**Transactional Data Structure Libraries (TiXeL)**

**Multi-Core Architecture and Systems**

**Ofir Yakovian**

1. **Introduction**

Designing a version of the **MCS** lock in which unlock completes in a constant number of steps (and thus, unlock completion never depends on the progress of another thread), must revise **Line 27**:

In our modern world, where multi-core architectures are the dominant computing platforms,

concurrent data structures play a major role in a lot of algorithms. Concurrent data

structures are signi\_cantly more di\_cult to design and implement than their sequential

counterparts. Therefore, they usually only export a small number of operations that are

promised to be executed atomically. Most algorithms, however, require a composition

of many such operations - leaving programmers the need to thoroughly think about the

correctness of their concurrent implementations.

Transactional Data Structures are concurrent data structures that enable a composition of

any number of operations - which is promised to be executed atomically. A well-designed

Transactional Data Structures implementation should allow transactions to execute fast

with as few aborts as possible, while also maintaining the speed of single operations as

provided by classic Concurrent Data Structure Libraries (CDSLs).

**We introduce transactions into libraries of concurrent data**

**structures; such transactions can be used to ensure atomicity**

**of sequences of data structure operations. By focusing**

**on transactional access to a well-defined set of data structure**

**operations, we strike a balance between the ease-ofprogramming**

**of transactions and the efficiency of customtailored**

**data structures. We exemplify this concept by designing**

**and implementing a library supporting transactions**

**on any number of maps, sets (implemented as skiplists), and**

**queues. Our library offers efficient and scalable transactions,**

**which are an order of magnitude faster than state-of-theart**

**transactional memory toolkits. Moreover, our approach**

**treats stand-alone data structure operations (like put and enqueue)**

**as first class citizens, and allows them to execute with**

**virtually no overhead, at the speed of the original data structure**

**library.**

**Data structures are the bricks and mortar of computer programs.**

**They are generally provided via highly optimized libraries. Since the advent of the multi-core revolution, many**

**efforts have been dedicated to building concurrent data**

**structure libraries (CDSLs) [1, 2, 5, 9, 11, 13, 21, 25, 27,**

**28, 36, 37, 44, 48, 49], which are so-called “thread-safe”.**

**Thread-safety is usually interpreted to mean that each individual**

**data structure operation (e.g., insert, contains, push,**

**pop, and so on) executes atomically, in isolation from other**

**operations on the same data structure.**

**Unfortunately, simply using atomic operations is not always**

**“safe”. Many concurrent programs require a number**

**of data structure operations to jointly execute atomically, as**

**shown in [47]. As an example, consider a server that processes**

**requests to transfer money to bank accounts managed**

**in a CDSL. If several threads process requests in parallel,**

**then clearly, atomicity of individual CDSL get and insert**

**operations does not suffice for safety: two concurrent**

**threads processing transfers to the same account may obtain**

**the same balance at the start of their respective operations,**

**causing one of the transfers to be lost.**

**This predicament has motivated the concept of memory**

**transactions [24] spanning multiple operations, which**

**appear to execute atomically (all-or-nothing) and in isolation**

**(so no partial effects of on-going transactions are observed).**

**A transaction can either commit, in which case all**

**of its updates are reflected to the rest of the system, or**

**abort, whereby none of its updates take effect. Transactions**

**have been used in DBMSs for decades, and are broadly**

**considered to be a programmer-friendly paradigm for writing**

**concurrent code [22, 46]. Numerous academic works**

**have developed software transactional memory (STM) toolkits**

**[10, 26, 43]. Moreover, some (limited) hardware support**

**for transactions is already available [33].**

**Nevertheless, as of today, general-purpose transactions**

**are not practical. STM incurs too high a overhead [6] and**

**hardware transactions are only “best effort” [33]. And in**

**both cases, abort rates can be an issue. Thus, with the exception**

**of eliding locks [45] in short critical sections, transactions**

**are hardly used in industry today. CDSLs, despite**

**their more limited semantics, are far more popular. Efficient**

**CDSL implementations are available for many programming**

**languages [1, 2, 36, 37] and are widely adopted [47].**

**1.2 Contributions**

**Our goal in this paper is to provide transaction semantics**

**for CDSLs without sacrificing performance. We introduce**

**in Section 2 the concept of a transactional data structure**

**library (TDSL), which supports bundling sequences of data**

**structure operations into atomic transactions. Individual operations**

**are seen as singleton transactions (singletons for**

**short). TDSLs provide composability; for example, a transaction**

**may invoke operations on two different maps and a**

**queue. But unlike STM approaches, atomicity only encompasses**

**the TDSL’s operations, whereas other memory accesses**

**are not protected.**

**Restricting the transactional alphabet to a well-defined**

**set of operations (e.g., enqueue, dequeue, insert, remove,**

**and contains) is the key to avoiding the notorious overhead**

**associated with STM.We show that we can benefit from this**

**restriction in three ways:**

**1. First, while a TDSL implementation may use standard**

**STM techniques, it can also apply CDSL-like customtailored**

**optimizations that rely on the specific data structure’s**

**semantics and organization in order to improve efficiency**

**and reduce the abort rate. For example, it can**

**employ STM-like read-set tracking and validation [10],**

**but reduce the read-set size to include only memory locations**

**that induce real semantic conflicts. Another example**

**is to use transactional access to a core data structure**

**that ensures correctness but does not support fast lookup,**

**and complement it with a non-transactional index for fast**

**lookup.**

**2. Second, a TDSL can employ different STM strategies**

**for managing different data structures within the same**

**library. For example, transactional access to maps is**

**amenable to optimistic concurrency control, since operations**

**in concurrent transactions are unlikely to conflict.**

**But when queues are used inside transactions, contention**

**is frequent, and so a pessimistic solution is often more**

**efficient. A TDSL can combine the two, by using optimistic**

**concurrency control for its maps and a pessimistic**

**approach for its queues.**

**3. Third, a TDSL can treat singletons as first class citizens**

**– it can spare them the transaction management overhead**

**altogether, and save programmers the need to deal with**

**their aborts.**

**We exemplify these three ideas in Section 3, where we**

**present example TDSL algorithms for popular data structures**

**– maps and sets (implemented as skiplists), and queues**

**– as well as compositions thereof. In Section 4 we generalize**

**this concept, and discuss a generic approach for composing**

**TDSLs with each other as well as with STM toolkits such as**

**TL2 [10]. Such a composition can provide, on the one hand,**

**high performance transactions comprised of data structure**

**operations, and on the other hand, fully general transactions,**

**including ones that access scalars.**

**We implement our new algorithms in C++. Our evaluation**

**in Section 5 shows that we can get ten-fold faster**

**transactions than STMs in update-dominated workloads accessing**

**sets, and at the same time cater stand-alone operations,**

**(i.e., singletons), on par with state-of-the-art CDSLs.**

**We further use our library to support Intruder [19] –**

**a multi-threaded algorithm for signature-based network intrusion**

**detection – and it runs up to 17x faster than using a**

**state-of-the-art STM.**

**Finally, we note that our example TDSL is by no means**

**exhaustive. Our goal here is to put forth TDSL as a new concept**

**for concurrent programming, which can offer programmers**

**the ease-of-use of transactions at the speed of CDSLs.**

**Section 6 compares this paradigm with earlier ideas in the**

**literature. We conclude in Section 7 by expressing hope that**

**the community will adopt this new concept and build additional**

**TDSLs.**

CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice. The suite of CMake tools were created by Kitware in response to the need for a powerful, cross-platform build environment for open-source projects such as ITK and VTK.

**While TiXeL is an open-source, cross-platform foundation, designed to build, test, and package across OS retails – its compilation process is (currently) bound to MSVC.**

**To use MSBuild.exe, one should invoke the BuildProject.bat on a Windows10 device. That will, in turn, include several CL switches to specify various aspects of our compilation process.**

**Every switch is available in two forms: -switch and /switch. The documentation only shows the -switch form. Switches are not case-sensitive. If you run MSBuild from a shell other than the Windows command prompt, lists of arguments to a switch (separated by semicolons or commas) might need single or double quotes to ensure that lists are passed to MSBuild instead of interpreted by the shell.**