**Boost Library: Lock Free Transactional Transformation**

**Members:** Sasuke Hirano, Christofer Padilla, Richard Snyder, Hugens Ulysse

**Sponsor:** Dr. Damian Dechev

**Group 4**

**Table of Contents**

Executive Summary 1

Project Significance 2

Blahblahblah x

y

z

**Executive Summary**

In this project, Group 42 will be conducting research and development on the code base provided by Dr. Damian Dechev and his Ph.D. students on Lock Free Transactional Transformation. Lock Free Transactional Transformation, or LFTT, is a library of concurrent data structures that focuses on linked base data structures.

This project has three main goals: to refactor the existing code, to conduct regression testing on the library to correlate with the Boost Library community standards, and to implement LFTT into current architectures that could utilize LFTT’s inherent qualities on linked data structures, such as linked lists and maps.

This project was started by Dr. Dechev’s lab, otherwise known as Area67. The team is currently focusing on concurrency, which is a field of computer science that exists to harness the capabilities of multicore processors. As computer chips near the physical limit of size, computer engineers are moving towards solutions of multiple logical processors for computer chips. However, this means that algorithms must be changed and updated in order to properly utilize and maintain correctness across multiple cores. The significance of this project will be relevant in the entirety of the future of computer science and engineering, as more and more libraries implement concurrent implementations of their existing technologies.

Concurrency is by no means an easy subject, nor is a graduate level project such as LFTT. Thus, throughout the duration of the project, Dr. Dechev will be instructing the group through bi-weekly sprint projects that are based around concurrency and the LFTT code base. The completion of these tasks will give the team a graduate level understanding of the topics at hand.

Implementing LFTT into concurrent capable architectures will begin with the in memory database created by MIT, known as Silo; this will be done using the *C++* language. Benchmarking and analysis of Silo utilizing LFTT will be conducted, with respect to other concurrent libraries as well as sequential models. Proving the efficiency of LFTT over current implementation methods is a key goal.

**Project Significance**

There exists in computer science and engineering Moore’s Law, which states that the amount of transistors on a computer chip will double every eighteen to twenty four months. However, computer engineers are reaching the fabricated lower limits of physical space between transistors that is possible; transistors currently sit about 10 nanometers apart from each other, which is still an absurdly small distance. However, most manufacturers and engineers believe that the distance of 7 nm is the physical lower limit that transistors can withhold without experiencing quantum tunneling. Because of this phenomenon, should engineers simply give up and accept that Moore’s Law will end once they reach 7nm?

Of course not! The solution that has been presented in the computing community is not to simply make the individual processors smaller, but to make more processors of that significantly small size and have them work in tandem. Multicore processors are chipsets that contain more than one logical processing unit with which all instructions of the chipset architecture can be run. These types of processors dawned in the early 2000’s when Intel and AMD created their first multicore chipsets, the Intel Core Duo and the AMD 64 Athlon X2.

Transactions are the means by which multiple threads can conduct multiple instructions on data structures in shared memory, thus taking full advantage of multicore processors and thread technologies. Methodologies of how to implement those transactions, such as Software Transactional Memory (STM) and Transactional Boosting, have been effective in the past, but are slightly inefficient in their necessary overhead and additional computation. Thus, Dr. Damian Dechev and his team of Ph.D students created Lock Free Transactional Transformation, or LFTT, a methodology that extends the characteristics to STM and Transactional Boosting to become a more efficient way of utilizing multiple cores on linked data structures.

In this project, we will attempt to allow show the usefulness of LFTT in architectures that can take advantage of multicore processors and linked data structures, such as in-memory databases. We will also refactor the original codebase that Dr. Dechev’s team created in order to submit that code base to the Boost Library, which is a collection of *C++* libraries that have the potential to be implemented into the standard language. In doing so, we will conduct regression tests on LFTT against current methodologies of concurrency and in-memory database manipulation to prove LFTT’s improvements over current implementations.

LFTT could potentially be implemented in the next release of *C++,* which could lead to significant improvements in the concurrent community and how programs are run. By taking full advantage of the advancements engineers have made in computing power, Moore’s Law can be preserved as well as allowing processes to continue to improve in efficiency and ability.

**Personal Statements**

**Sasuke Hirano**

**Christofer Padilla**

**Richard Snyder**

This project has a lot of significance to me as I move forward towards pursuing a graduate degree in computer science. Advanced topics such as concurrency are necessary for a mastered understanding of computer science, so taking on this project will jumpstart my further education and give me a head start on my graduate peers. Coming into the Computer Science program with no prior programming experience was a daunting task; however, seeing how far I’ve come is extremely uplifting, especially in terms of my willingness to take on such an advanced topic as the project that determines my graduation status. I have had prior experience to *C++* before this project, but the libraries that will be implemented in LFTT that include locks and compare and swaps are completely new to me. The challenging part of this project will most certainly be building the data structures that Silo can run its operations on. The underlying code base for running LFTT is already in place, but implementing that code into an existing architecture is something that will always be a daunting task. I foresee my team and I having to spend serious time and devotion to the project, but in the end we will end with a fantastic story to tell, as well as a reference to a world-renowned professor and researching in Dr. Dechev.

**Hugens Ulysee**

I chose the AREA67 project with Dr. Dechev because I noticed this project involved concurrency in *C++*. A few weeks prior to our project selection, I was speaking to a friend about programming topics and I mentioned my lack of understanding in the topic of concurrency. My friend suggested a book for me to read, but shortly after that conversation, our Senior Design class was presented with the AREA67 project, so I figured this project would be a perfect way for me to get an understanding of concurrency. Although this project was not in my top 3 preferences for projects to work on, I’m glad I was placed on this project, as I am now gathering a better understanding of concurrency through a hands-on learning experience. In addition to learning concurrency, I’m also learning *C++*, which I also had no prior experience with. Due to the fact that my experience with these topics seem to be far less than that of my team members, I completely understand that I’ll face a few challenges and I’ll undergo learning curves; regardless of those facts, I’m motivated to provide my gained knowledge to the benefit of the group. Aside from my personal challenges, I’m starting to see a few challenges that may present themselves to our group as a whole. The first challenge would be enabling LFTT to run operations on multiple data structures in one shared transaction. Another challenge is implementing LFTT with the Silo database. Although we have a general idea of how to implement the two, it will be a challenge to actually do so. Nevertheless, I’m extremely excited to continue my research and my learning of these topics and I’m excited to collaborate with my team members to provide a working solution to the problem we were proposed by Dr. Dechev.

**Goals/Requirements**

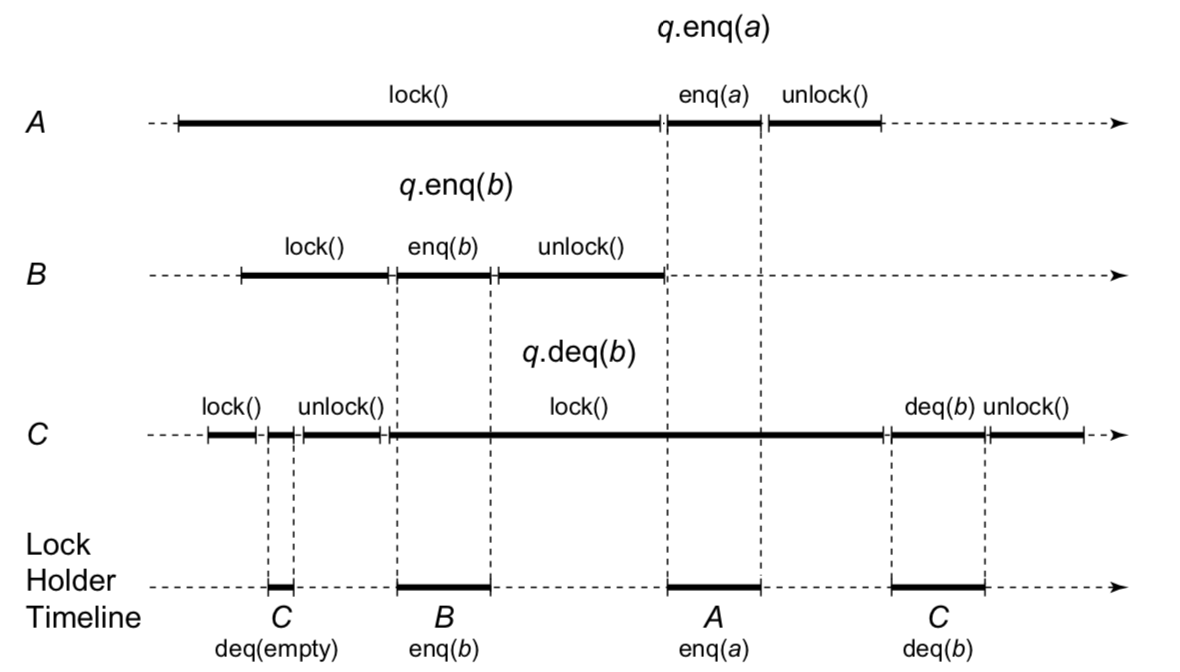
**Specifications**

**Research**

**Concurrent Objects**

Concurrent objects are simply objects (threads) created in programming where they are executed simultaneously. These objects are described through the properties of correctness and progress. The correctness can be best defined as the the specification and verification of what a given program actually does. There are three main conditions for correctness: quiescent condition, sequential consistency, and linearizability. Progress can also be referred to as the liveness property ensuring there is forward movement in the program execution. It is simply categorized into blocking and nonblocking method implementations.

**Correctness**

Correctness can be explained more simply through an example of a lock-based concurrent FIFO queue. Suppose there are enqueue and dequeue methods which synchronize by a mutual exclusion lock (mutual exclusion describes the problem where only one thread can execute a particular segment of code). This can be seen as a correct concurrency FIFO queue due to the nature of sequential execution where accessing and updating fields occur while holding a lock state. As an example, a timeline can be used to describe the locking queue execution.

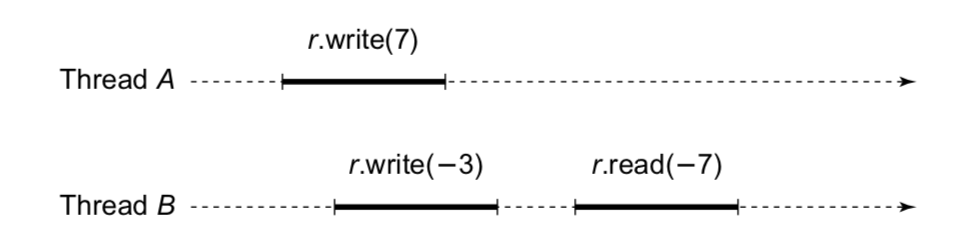
C initially acquires the lock and the attempts to dequeue from the empty queue and releases the lock. It then throws an exception since C tried dequeuing from and empty queue. During this period, B starts the process of acquiring the lock and actually acquires it soon as C releases the lock. Then, it inserts (enqueue) b and releases the lock. A finally acquires the lock and inserts a and releases the lock. C will re-acquire the lock, dequeues b, then releases the lock and the execution will complete as it returns.

**Sequential Objects**

An object can be simply described as a container for data in C++ where the methods provided by the object manipulate the object themselves. These objects also have a class defining these methods and how they behave. The application program interface (API) documentation typically will provide the information that describe the states of the object before and when the method returns. It divides this process into precondition and postcondition states defining sequential specification which is the style of documentation. This takes place when a single thread manipulates a collection of objects.

Sequential specification documents falls apart when objects are shared by multiple threads. This is due to the nature of an object’s method being invoked by concurrent threads where the method calls can overlap in time and the order no longer make sense. If two variables are enqueued on a FIFO queue during overlapped intervals, it becomes confusing to describe the states of the objects.

**Quiescent Consistency**

A method call is defined as the internal that starts with an invocation event and ends with a response event. Method calls by concurrent threads may overlap whereas a single thread method call is sequential. These method calls are pending when the events occur but the response event has not occurred.

The method calls above is an example of two calls taking place instantaneously. The register r represents an object version of a read-write memory location. Two threads concurrently write -3 and 7 to a shared register r. Thread B reads the value -7 and returns. The return value is not the combination of the two but rather a single value of the two.

This is where the quiescent object is introduced. It simply takes place when an object has no pending method calls. Method calls should appear to happen in sequential order one-at-a-time.

Method calls separated by a period of quiescence should appear to take effect in their real-time order. These two principles define a correctness property called quiescent consistency. For example, suppose A and B concurrently enqueue x and y in a FIFO queue. The queue becomes quiescent, and then C enqueues z. We may not be able to predict the relative order of x and y in the queue, but we know they are ahead of z. Informally, it says that any time an object becomes quiescent, then the execution so far is equivalent to some sequential execution of the completed calls.

**Sequential Consistency**

The order in which a single thread issues method calls is called its program order. (Method calls by different threads are unrelated by program order.)

Method calls should appear to take effect in program order. Sequential consistency requires they occur in a sequential order consistent with program order or at least act to do so. In any concurrent execution, there is a way to order the method calls sequentially so that they are consistent with program order, and meet the object’s sequential specification. There may be more than one order satisfying this condition. There are two possible sequential orders that can explain how thread A enqueues x while B enqueues y, and then A dequeues y while B dequeues.The order occurs so that: A enqueues x, B enqueues y, B dequeues x, then A dequeues y, OR B enqueues y, A enqueues x, A dequeues y, then B dequeues x. Both these orders are consistent with the method calls’ program order, and either one show sequentially consistency.

**Concurrency and Linearizability**

The principal drawback of sequential consistency is that it is not compositional: the result of composing sequentially consistent components is not itself necessarily sequentially consistent. We propose the following way out of this dilemma. Let us replace the requirement that method calls appear to happen in program order with the following stronger restriction where method calls should appear to take effect instantaneously at some moment between the invocation and response. This states that the real-time behavior of method calls must be preserved. We call this correctness property linearizability. Each linearizable execution is sequentially consistent, but not vice versa.

The most common and recommended way to show that a concurrent object implementation is linearizable is by expressing a linearization point for each method where the method takes effect. Each method’s critical section can serve as its linearization point for lock-based implementations. The linearization point is typically a single step where the effects of the method call become visible to other method calls for implementations that do not use locking,

**Progress Conditions**

Blocking is defined when an unexpected delay by one thread can prevent others from making process. The machine and the operating system dictate how theses delays occur and are handled. A wait-free method is defined when it finishes execution in a finite number of steps. There is also an unbounded wait-free method when there is a limit to the steps a method call can take. Where there is no limitation on the number of threads will be then described as population-oblivious method.

To reiterate, the key difference between wait-free and lock-free algorithms is that wait-free guarantees threads make progress whereas the lock-free guarantees a method call will finish in a finite number of steps. This indicates that any wait-free algorithms are also lock-free but no vice versa.

Another condition is to have a obstruction-free method that finished in a finite number of steps from any instance after the isolation execution.

These conditions depend on the need of the program whether which condition to apply to the implementation of concurrent objects. These dependent properties will bring simplicity and efficiency to the applications.

**Locking with Linked Lists**

…

**Boost C++**

**Prerequisites**

The information on Boost requires the basic foundation of programming quality programming practices.

**Short Background on Boost**

Boost is essentially a set of open source libraries for the C++ programming language. It is a complement to the STL (standard template library) used by C++ and offer advanced tasks and structures used in multithreading, linear algebra, unit testing, and more. Some of the reasons for the popularity of Boost includes the open-source accessibility and the use of peer-reviewed code. Some of the frequently used features of Boost will actually get pushed to the STL depending on the popularity. Another reason why Boost is useful is because the features are rarely dependent on each other making it less complex to use.

**Reason for Submission to Boost**

Libraries that are submitted and accepted into Boost have high-visibility and high-impact. Many major companies such as CERN and Google use Boost as it provides libraries that are elegant and efficient. It speeds up initial development resulting in fewer bugs and cuts long-term maintenance costs. Although the actual submission of the library is outside the scope of this project, it adds tremendous experience of understanding and gaining knowledge of what the submission standards are and also the process of doing so.

**Getting Started**

Actual Boost library requirements and guidelines are posted on <https://www.boost.org/development/requirements.html>. This documentation is provided to ensure the universal guidelines for submitting a library to Boost. There is also a submission process page on <https://www.boost.org/development/submissions.html>.

**Boost Library Requirements and Guidelines**

**Requirements 1.1**

Proposed libraries not within specs of Boost Requirements will be immediately rejected.

* The license must meet the license requirements below. Restricted licenses like the GPL and LGPL are not acceptable.
* The copyright ownership must be clear.
* The library should be generally useful.
* The library must meet the portability requirements below.
* The library should preferably meet the organization requirements below. But is only required to meet them after acceptance.
* The library must come reasonably close to meeting the Guidelines below.
  + Design and Programming
  + Filenames
  + Documentation
* The author must be willing to participate in discussions on the mailing list, and to redefine the library accordingly.

There's no requirement that an author read the mailing list for a time before making a submission. It has been noted, however, that submissions which begin "I just started to read this mailing list ..." seem to fail, often embarrassingly.

**License Requirements 1.1.1**

* Must be simple to read and understand.
* Must grant permission without fee to copy, use and modify the software for any use (commercial and non-commercial).
* Must require that the license appear on all copies of the software source code.
* Must not require that the license appear with executables or other binary uses of the library.
* Must not require that the source code be available for execution or other binary uses of the library.
* May restrict the use of the name and description of the library to the standard version found on the Boost web site.

**Portability Requirements 1.1.2**

* A library's interface must portable and not restricted to a particular compiler or operating system.
* A library's implementation must if possible be portable and not restricted to a particular compiler or operating system. If a portable implementation is not possible, non-portable constructions are acceptable if reasonably easy to port to other environments, and implementations are provided for at least two popular operating systems (such as UNIX and Windows).
* There is no requirement that a library run on C++ compilers which do not conform to the ISO standard.
* There is no requirement that a library run on any particular C++ compiler. Boost contributors often try to ensure their libraries work with popular compilers. The boost/config.hpp configuration header is the preferred mechanism for working around compiler deficiencies.

*The demonstration of compiling and executing correctly with two different C++ compilers, often under different operating systems indicate the proof of portability.*

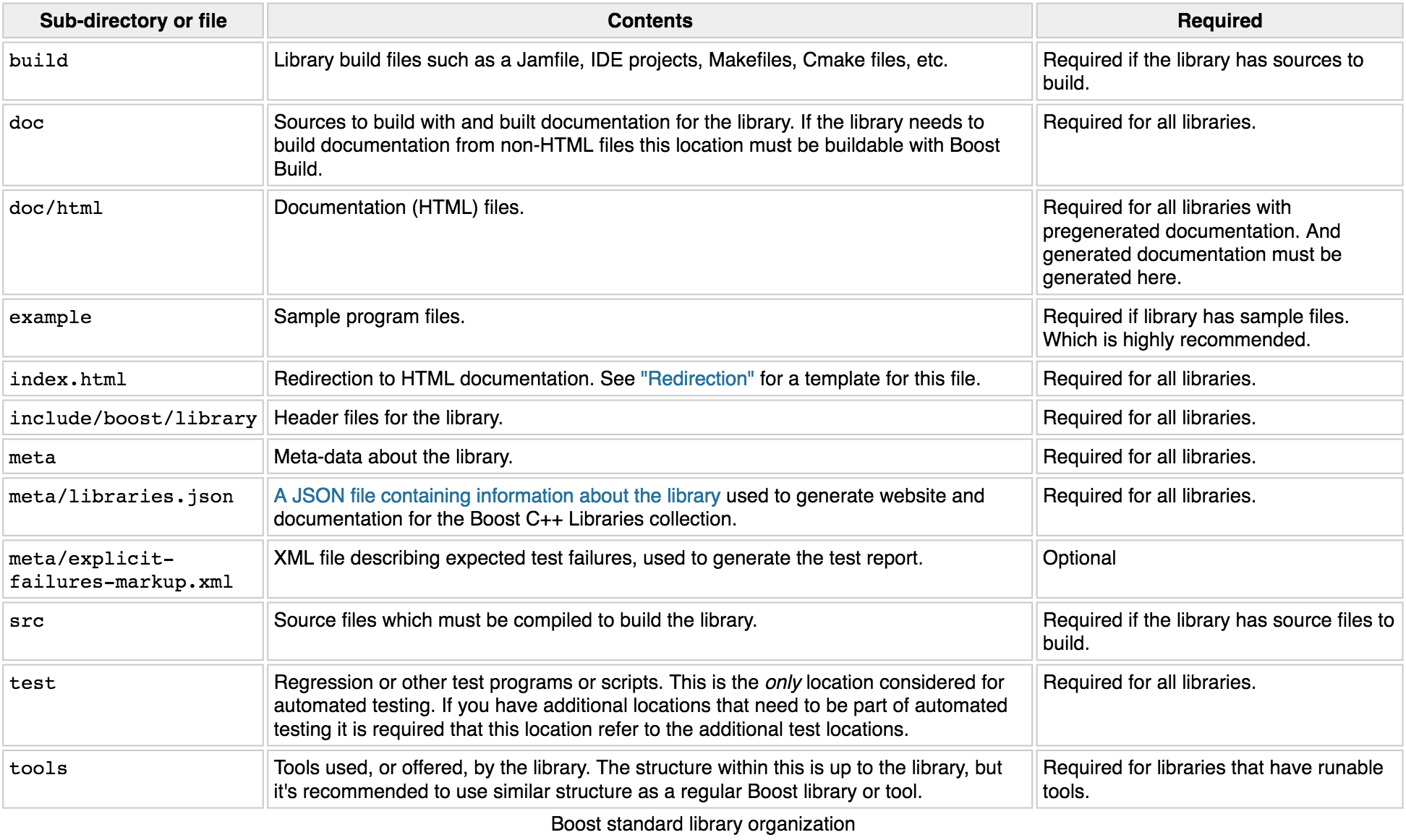
**Ownership 1.1.3**

According to “How to Copyright Software” by MJ Salone, Nolo Press, 1990 says:

Doing work on your own time that is very similar to programming you do for your employer on company time can raise nasty legal problems. In this situation, it's best to get a written release from your employer in advance.

Copyright notice in all important files are required for submission. Boost does not accept libraries without clear copyright information.

**Organization 1.1.4**

The entirety of organizing a consistent view of the libraries to users are just as important as the quality of the APIs and code design of Boost. Upon acceptance, libraries must adhere to this directory and file structure:

**Integration 1.1.5**

Once a library is accepted into Boost as part of its libraries, like most software development integration requires proper development, testing, documentation, and release processes. This will increase the overall quality expected by its users of which the integration follow as:

***Building Sources 1.1.5.1***

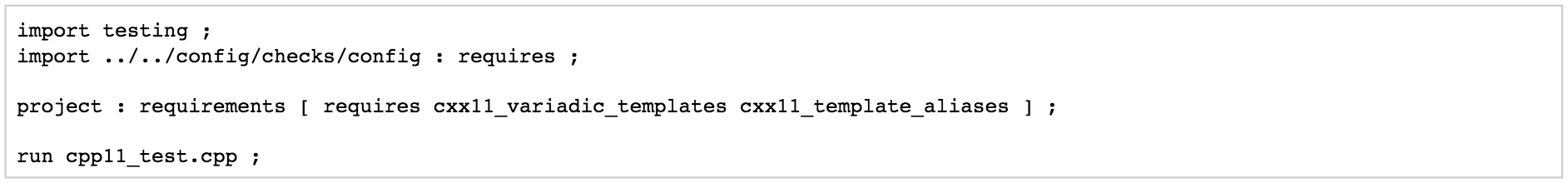
The submission library needs to provide a Boost Build project that the user, and the top level Boost project, can use to build the library if it has sources to build. The Jamfile for the source build needs to minimally declare the project, the library target(s), and register the target(s) for installation. For example:

***Testing 1.1.5.2***

The library needs to provide a Boost Build project that the user, and the root Boost test script, can use to build and run the tests for the library. The testing build project must reside in the project-root/test directory and must be buildable from this or another directory (for example, b2 libs/library/test from the Boost root must work.)

An example test/Jamfile is given below:

*WARNING:* This is the only location considered for testing by the top level testing script. If you want to test additional locations you must declare such that they are built as dependencies or by using build-project.

If the library requires a level of C++ conformance that precludes certain compilers or configurations from working, it's possible (and recommended) to declare these requirements in the test Jamfile so that the tests aren't run, to conserve test resources, as given in the example below:

***Building Documentation 1.1.5.3***

The library needs to provide a Boost Build project for building the documentation for the library. The project-root/doc project is the only location refered to by the top level documentation build scripts and the release building scripts. The documentation build project must have the following two features:



1. Define a boostdoc target. This target should likely be an alias that looks roughly like:

But if your project doesn't integrate into the global documentation book you can use an empty alias like:

1. The project must default to building standalone documentation if it has any. The release scripts build this default so as to guarantee all projects have up to date documentation.

**Guidelines 1.2**

Guidelines provide a form of checklist for preparing the content of a library submission. Every guideline will not apply interchangeably through all libraries but a reasonable effort to comply is expected.

**Backwards Compatibility 1.2.1**

Library authors are encouraged to follow a few guidelines before introducing breaking changes to the library being submitted. This ensures a more safe transition from old APIs to new ones.

1) Non-breaking changes can be done without restriction

2) Users should be notified a few releases before the change is published for small breaking changes. Most breaking changes fall into this category.

3) For large breaking changes with a migration path from the old API to the new API (for example boost::filesystem v2 to v3), the new API should be introduced in a separate directory/namespace, and users should be noticed and given a few releases to move over. The old API can be removed after some time.

4) For large breaking changes without a migration path (for example boost::spirit v2 to v3), the new API should be provided in a separate directory/namespace, and the old API should be preserved (because there's no migration path). Removing the API should be considered the same as removing a Boost library, which can be done but needs a more extensive deprecation period.

5) Large breaking changes that are equivalent to a redesign or rewrite of the library should be treated as a new library and a formal review (or at least a mini review) is encouraged.

**Design and Programming 1.2.2**

The priority for designing and programming a library is having clarity and correctness with optimization coming after. This starts with utilizing the STL and avoiding non-standard compiler extensions.

Headers should be “good neighbors” (header files co-existing peacefully and productively with the users’ code and their libraries). This also follow with naming consistency.

* Names (except as noted below) should be all lowercase, with words separated by underscores.
* Acronyms should be treated as ordinary names (e.g. xml\_parser instead of XML\_parser).
* Template parameter names begin with an uppercase letter.
* Macro (gasp!) names all uppercase and begin with BOOST\_.

Please review the Design and Programming section of the Boost Library Requirements and Guidelines page at <https://www.boost.org/development/requirements.html#Design_and_Programming> for information regarding implementation variation, the Best Practices Handbook, and guidelines for libraries with separate source to see how to ensure that compiled link libraries meet user expectations.

Avoid exception-specifications but do use exceptions to report errors where appropriate, and write code that is safe in the condition of exceptions.

Provide some sort of sample programs or confidence tests to show how the library is used. This will mean that a regression test program or programs need to be provided.

There are conservative guidelines for the use of font, tabs, and unrestricted line lengths in the code.

* Use fixed-width fonts. Reference fonts rationale .
* Use spaces rather than tabs. See [tabs rationale](https://www.boost.org/development/requirements.html#Tabs).
* Limit line lengths to 80 characters.

End all documentation files (HTML or otherwise) with a copyright message and a licensing message.

List of items required for the beginning of all source files:

* A comment line describing the contents of the file.
* Comments describing copyright and licensing: again, the preferred form is indicated in the license information page at <https://www.boost.org/users/license.html>.
* Note that developers should not provide a copy of LICENSE\_1\_0.txt with their libraries: Boost distributions already include a copy in the Boost root directory.
* Some of the Boost's automatic tools depend on a comment line referencing your library on the Boost web site identifying which library the header files belong to. For example:

If you want to add runtime assertions to your code (you should!), use Boost’s BOOST\_ASSERTmacro (in boost/assert.hpp ). Use BOOST\_ASSERT in public headers and in library source code (for separately compiled libraries). Use of C's assert macro is ok in examples and in documentation but not recommended.

The code needs to compile in the presence of the min() and max() macros. Some platform headers define min() and max() macros which may cause the failure C++ constructs compilation. Some simple tricks can protect the code from inappropriate macro substitution:

* If you want to call std::min() or std::max():
  + If you do not require argument-dependent look-up, use (std::min)(a,b).
  + If you do require argument-dependent look-up, you should:
    - #include <boost/config.hpp>
    - Use BOOST\_USING\_STD\_MIN(); to bring std::min() into the current scope.
    - Use min BOOST\_PREVENT\_MACRO\_SUBSTITUTION (a,b); to make an argument-dependent call to min(a,b).
* If you want to call std::numeric\_limits<int>::max(), use (std::numeric\_limits<int>::max)() instead.
* If you want to call a min() or max() member function, instead to doing obj.min(), use (obj.min)().
* If  you want to declare or define a function or a member function named min or max, then you must use the BOOST\_PREVENT\_MACRO\_SUBSTITUTION macro. Instead of writing int min() { return 0; } you should write int min BOOST\_PREVENT\_MACRO\_SUBSTITUTION () { return 0; } This is true regardless if the function is a free (namespace scope) function, a member function or a static member function, and it applies for the function declaration as well as for the function definition.



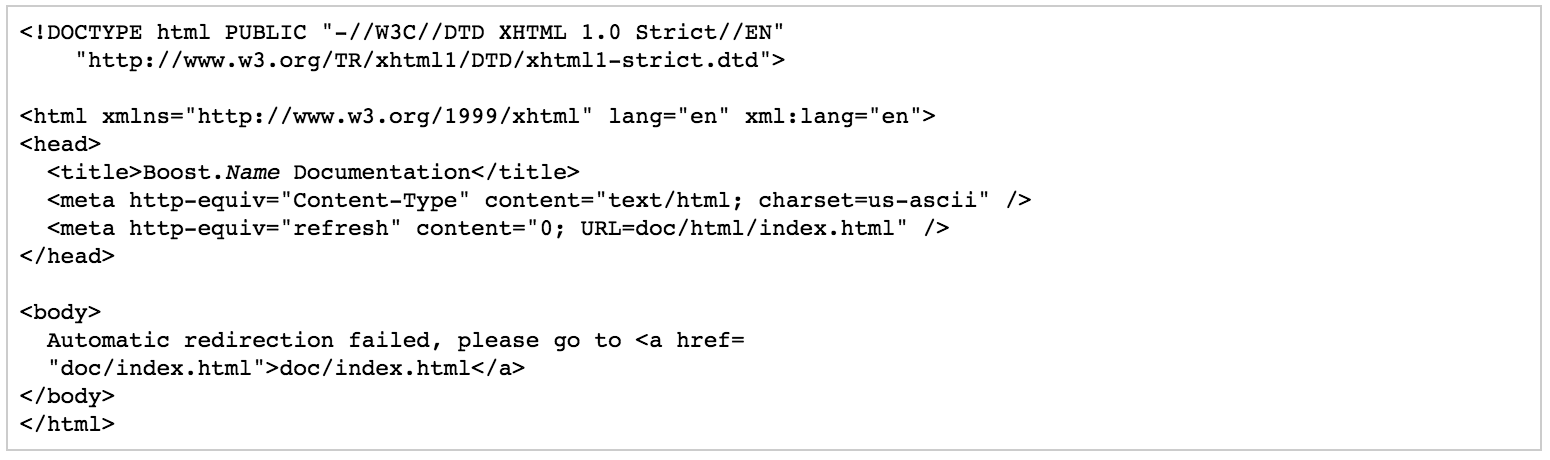
**Filenames 1.2.3**

This section is only intended to clarify naming conventions regarding files and directories. Use a three-letter filename extension ending with “pp” for files intended to be processed by a C++ compiler.

* Names must contain only lowercase ASCII letters ('a'-'z'), numbers ('0'-'9'), underscores ('\_'), hyphens ('-'), and periods ('.'). Spaces are not allowed.
* Directory names must not contain periods ('.').
* The first and last character of a file name must not be a period ('.').
* The first character of names must not be a hyphen ('-').
* The maximum length of directory and file names is 31 characters.
* The total path length must not exceed 207 characters.

***Redirection 1.2.3.1***

The primary directory should always contain a file named “index.html”. Authors have requested this so that they can publish URL's in the form *https://www.boost.org/libs/lib-name* with the assurance a documentation reorganization won't invalidate the URL. Boost's internal tools are also simplified by knowing that a library's documentation is always reachable via the simplified URL.

The primary directory index.html file should just do an automatic redirection to the doc/html subdirectory like so:

**Naming Consistency 1.2.4**

In addition to all the naming conventions descriptor throughout the Guidelines section (1.2), here are other library naming consistency that help the library developers and users with their experience.

* Cryptic abbreviations are strongly discouraged.
* Names are usually singular rather than plural. For example, "filesystem" is recommended in comparison to "filesystems", "fs" or "nicecode".
* The library's primary directory (in parent *boost-root/libs*) is given that same name. For example, *boost-root/libs/filesystem*.
* The library's primary header directory (in boost-root/libs/name/include) is given that same name. For example, boost-root/libs/filesystem/boost/filesystem.
* The library's primary namespace (in parent *::boost*) is given that same name, except when there's a component with that name (e.g.,*boost::tuple*), in which case the namespace name is pluralized. For example, *::boost::filesystem*.

Follow these conventions when documenting Boost libraries, follow these conventions also.

* The library name is set in roman type.
* The library name is capitalized.
* A period between "Boost" and the library name (e.g., Boost.Bind) is used if and only if the library name is not followed by the word "library".
* The word "library" is not part of the library name and is therefore lowercased.

A few examples of how to apply these conventions are listed below:

* Boost.Bind was written by Peter Dimov.
* The Boost Bind library was written by Peter Dimov.
* I regularly use Bind, a Boost library written by Peter Dimov.

**Documentation 1.2.5**

Documentation should be proportional to the need. The documentation should assume the readers have a basic knowledge of C++, but are not necessarily experts.

The format for documentation should be in HTML with the entry point being a file names index.html. It should not require an advanced browser or server-side extensions. Style sheets are acceptable. ECMAScript/JavaScript is discouraged. See section 1.2.3.1 for more information.

* General introduction to the library. The introduction particularly needs to include:
  + A very high-level overview of what the library is good for, and perhaps what it isn't good for, understandable even by those with no prior knowledge of the problem domain.
  + The simplest possible ("hello world") example of using the library.
* Tutorial covering basic use cases.
* Reference documentation:
  + Description of each class.
  + Relationship between classes.
  + For each function, as applicable, description, requirements (preconditions), effects, post-conditions, returns, and throws.
  + Discussion of error detection and recovery strategy.
* How to compile and link.
* How to test.
* Version or revision history.
* Rationale for design decisions.
* Acknowledgements.

**Boost Library Submission Process**

This section provides the library submission process to Boost. This is condensed into 10 steps which include in depth requirements and guidelines.

**Learn About Boost (Step 1) 2.1**

A recommended place to begin is by following posts on the main developers mailing list or looking through the archives. The Boost web site offer tons of information including the Requirements page described above. Search the web to get an idea of the commitment required to get a library into Boost. A whole Boost community exists aimed at encouraging high quality libraries by a process of discussion and refinement. Libraries make take several years to get it past review so a committed preparation is certainly required.

**Determine Interest (Step 2) 2.2**

It is recommended to participate in reviews for other submissions to become familiar with the process and the emotional demands of a formal review.

The atlas of libraries in Boost is enormous and is highly recommended to research before the designing of a library. It may already be in existence or another candidate may possibly be in development for a similar work. This also allows a chance to even collaborate which may be considered more efficient.

It is important to note that libraries are only reviewed when there is enough appeal to form a viable peer review. There are many places to gain traction such as the Boost developers mailing list, the Boost Library Incubator, or even in Reddit.

Posting long descriptions, documentation, or code to the mailing list should be avoided. It is recommended to post a web link that may have all this information.

**Start Development (Step 3) 2.3**

Making the library publicly available should only come if the response to an initial query indicates interest.

Code should be committed to a version control system such as Git with the documentation being available in HTML format on a public website such as Github.

The library should contain material as if it were on the [boost.org](http://boost.org) web site, It makes it easier for reviewers to copy code into the Boost distribution for testing.

As per one the requirements, verification of the library being able to compile under at least two compilers is needed.

Code should be released under the Boost Software License.

Submitting to the Boost Library Incubator is recommended to receive preliminary feedback and reviews.

**Refinement (Step 4) 2.4**

The cycle of refinement is to discuss, refine, rewrite.

The details will vary depending on the library but it usually happens publicly or on the mailing list. Frequent discussion happens in private emails. It can be either short or extend into longer periods. The archive of past messages is one way to see how this process worked for other Boost libraries. The *Best Practices Handbook* generally provide best practices with sample scripts and codes.

**Getting seconded for review (Step 5) 2.5**

When the library feels ready for entry into Boost, it is required to have at least one or more members of the Boost community to endorse the library for entry into Boost. A simple method of achieving this is to post to the Boost developers mailing list a short description of your library, links to its github and documentation, and a request for endorsements.

**Seek a Review Manager (Step 6) 2.6**

The author must find a volunteer (review manager) with knowledge of the library domain, and experience with the review process to schedule a formal review. Members can be found usually on the developer list who showed interest in the library.

Once a potential review manager has been identified, [contact the review wizards](https://www.boost.org/community/reviews.html#Wizard) for approval. The wizards approve review managers based on their level of participation in the Boost community. The review wizards will coordinate with both the author and review manager to schedule a date convenient for both.

See [Formal Review Process](https://www.boost.org/community/reviews.html) for further details along with the responsibilities of a review manager.

**Formal Review (Step 7) 2.7**

Every code example needs to be verified that it works, that all unit tests pass on at least two compilers on at least two major operating systems, and run your documentation through a spelling and grammar checker.

The master branch of the library must not be modified once the review process has started. A separate branch is recommended to be created so that adjustments can be made accordingly after reviews. For bigger ticket items of work, open issues on your issue tracker so interested people can track the fixing of specific issues raised.

The review manager will consider all the reviews made by members of the community and arrive at a decision on whether your library is rejected, conditionally accepted or unconditionally accepted. They will post a report summarizing the decision publicly. If conditions are attached to acceptance, you will need to implement those conditions or else undergo an additional formal review.

**Web site posting (Step 8) 2.8**

The submitter is usually given the Boost repository write access once an accepted library is ready for inclusion where the check-in and the maintenance of the library takes place. Any questions regarding the repository should be sent to the moderators of Boost.

**People Page (Step 9) 2.9**

A capsule biography and picture should be sent to the Boost webmaster if [boost.org](http://boost.org) does not have them on their website already. It is up to you as to whether or not the biography includes your email address or other contact information. The preferred image size is 500x375 with the format being .jpg but the webmaster has a photo editing software and can do the image preparation if necessary.

**Lifecycle (Step 10) 2.10**

Libraries are software; they lose their value over time if not maintained. Postings on the Boost developers or users mailing lists can alert you to potential maintenance needs; please plan to maintain your library over time. If you no longer can or wish to maintain your library, please post a message on the Boost developers mailing list asking for a new maintainer to volunteer and then spend the time to help them take over.

If there are any orphaned libraries, the Community Maintenance Team for Boost will handle any issues.

**Implementing Concurrency**

**Introduction**

The purpose of our implementation of the concurrent library as described in the paper *Lock-free Transactions without Rollbacks for Linked Data Structures* is to provide an improved methodology that does not rely on STM nor require the use of an additional data structure. The data structures in our library using the approach described in the paper provides substantial speedup over alternatives based on transactional boosting and the best implementations of STM. [1]

In this section, we will examine and compare some of the most popular libraries that provide tools for concurrency, most of them being popular STM libraries. In addition, we will also cover the extent of concurrency support different languages built into their standard libraries. Since our project aims to provide improved performance over STM, we will also cover some of the downsides and criticisms of transactional memory.

These are the topics we will examine:

# Libraries

* LibCDS
* Tervel
* RSTM
* Boost

# Languages

* Haskell
* Clojure

Before we examine them, however, we will talk about common features among concurrent libraries in general.

## Types of Non-Blocking Data Strucutres

The book *The Art of Multiprocessor Programming* is the first comprehensive presentation of the principles and tools available for programming multiprocessor machines and in it, Herlihy Shavit distinguish between 3 types of non-blocking data structures, each having different properties: [2]

* data structures are **wait-free**, if every concurrent operation is guaranteed to be finished in a finite number of steps. It is therefore possible to give worst-case guarantees for the number of operations.
* data structures are **lock-free**, if some concurrent operations are guaranteed to be finished in a finite number of steps. While it is in theory possible that some operations never make any progress, it is very unlikely to happen in practical applications.
* data structures are **obstruction-free**, if a concurrent operation is guaranteed to be finished in a finite number of steps, unless another concurrent operation interferes.

Our library contains lock-free transactional linked data structures and most of the data structures contained in the libraries we will study will also be lock-free.

## Safe Memory Reclamation

Any lock-free algorithm containing code of the form

Node\* currentNode = this->head; // assume the load from "this->head" is atomic

Node\* nextNode = currentNode->next; // assume this load is also atomic

must also deal with the problem of managing it’s own memory, also known as garbage-collecting (GC). Even if a language provides GC, no known GC is lock-free (let alone wait-free). This has been an area of much research since the advantages of designing or implementing a lock-free or wait-free data structure are nullified if you end up having to add a blocking mechanism for reclaiming memory. Thus it is important to have safe memory reclamation (SMR) that does not impede the performance of the data structures operations. Therefore, whatever memory reclamation method is used, it needs to have at least the same progress guarantees as the data structure itself. [3]

In addition to guaranteeing progress, SMR is also at least as important as the concurrent algorithm or data structure itself for three other reasons. First is performance. Just as it does not make sense to use a lock-free algorithm only for it to become blocking when garbage collecting happens, neither should we use a fast lock-free data structure only for it to be slowed by memory reclamation.

Second is simplicity. Since implementing a concurrent data structure itself might be complicated enough as it is, we do not want to have to risk compromising the structure with the addition of a garbage collector with a lot of overhead. The simpler the SMR, the less likely the user is to introduce mistakes or errors into the design.

Another advantage using a lock-free/wait-free data structure provides is that they are resistant to failure. This is especially important when talking about a concurrent system, because whether the concurrency happens across threads or server nodes, should one fail or go offline, the other processes accessing some shared data structure will continue to progress. For the most part, blocking memory reclamation does not have this property and adding one to a lock-free/wait-free data structure negates this advantage since once one process goes down, further memory reclamation will fail, eventually compromising the entire system.

There are three types of memory reclamation: atomic reference counting, quiescent-based, and pointer-based.

Atomic reference counting uses atomics to count each reference, similar to the way smart pointers do. The downside to this approach is that they are typically slower since we wave to traverse a list of nodes in order to increment and decrement counters.

Quiescent-based, where quiescent states are used to allow several copies of data to exist in memory, removing old copies when they’re no longer in use. This type includes Epoch-Based-Reclamation (EBR) and all Userspace RCUs and is the fastest memory reclamation (for readers). The downside is reclamation is inherently blocking.

Finally, we have Pointer-based reclamation, which includes Hazard Pointers, Hazard Eras, Pass The Buck, and Drop the Anchor. This type can be completely wait-free for readers and reclaimers, making them the only type capable of providing lock-free/wait-free progress guarantees. Their main problem is that they’re slow, though not as slow as atomic reference counting.

# Libraries

## LibCDS

# **Overview**

The Concurrent Data Structures (CDS) library is a collection of lock-free and lock-based fine-grained algorithms of data structures like maps, queues, lists, etc developed by Maxim Khizhinsky and other contributors in C++ 11. The CDS library provides several implementations of each of it’s lock-free data structures based on published papers. While each of these implementations use different memory reclamation schemas, they use a common interface for the type of data structure being used. Another advantage offered by this library is that it does not require a specific threading model to be used, as it’s developed to support multiple.

The library also provides different SMR algorithms to the user, including M. Michael’s Hazard Pointer, User-space Read-Copy Update (URCU), as well as an empty “GC” (no garbage collector) for append-only containers that do not support item reclamation.

# **Usage**

The main part of lock-free programming is SMR and usage of the library usually expects the application use one type of garbage collector. First, the cds library and the garbage collector are initialized in the main function. Second, any threads should be attached to the cds infrastructure. Afterwards, the cds lock-free containers can be used safely without any external synchronization.

# **Data Structures Provided**

LibCDS contains implementations of the following containers:

* lock-free stack with optional elimination support
* several algo for lock-free queue, including classic Michael & Scott algorithm and its derivatives, the flat combining queue, the segmented queue.
* several implementation of unordered set/map - lock-free and fine-grained lock-based
* flat-combining technique
* lock-free skip-list
* lock-free FeldmanHashMap/Set Multi-Level Array Hash with thread-safe bidirectional iterator support
* Bronson's et al algorithm for fine-grained lock-based AVL tree

LibCDS offers an intrusive and non-intrusive (STL-like) version of each container, separated under respective namespaces.

# **Dependencies**

* boost.thread
* boost.system
* google-test

For Windows builds, perl and boost 1.5 and above are also required.

GCC and Clang compilers are supported on Unix-like systems.

## Tervel

# **Overview**

Tervel is a wait-free framework and library of concurrent algorithms and containers developed by AREA67 and implemented in C++ 11. Its primary goal is to ensure that each component of Tervel is wait-free. [5]

Tervel is unique in that it is a framework for developing non-blocking, and more specifically wait-free, algorithms. It was developed in order to bring many techniques and methodologies found in literature into a single usable framework. The design of the framework is heavily influenced by the challenges and requirements the team faced when implementing wait-free algorithms.

# **Memory Reclamation**

To handle memory reclamation, Tervel uses thread-local and global memory pools. [6] It provides a comprehensive interface by which developers can use either hazard pointers or reference counting to ensure objects ar not reused or freed while a thread is operating on them.

# **Descriptor**

One technique used in non-blocking algorithms to allow progress to be made between threads is the descriptor object. A descriptor object is used to keep track of current pending operations. Each thread performing an operation will place a reference to a descriptor object into shared memory. The descriptor contains enough information about it’s operation that any other thread performing another operation can help the finish the pending operation described in the descriptor.

Tervel provides an abstract descriptor class. The onus is on the user to provide their own implementation of the value() and complete() member functions. The value() is calculated as either the logical value that the descriptor object replaced or, if the descriptors operation has been completed but the descriptor object itself has not yet been removed, the value determined by the operation that placed the descriptor. [6] The complete() function is used to remove a descriptor object from an address when it’s operation has completed.

This interface simplifies the complexity of linearizing an algorithm as the developer only needs to consider the case where a thread calls the complete() function of a descriptor object at an arbitrary point in time. So long as the algorithm’s operations are linearizable, then it can be shown that when two descriptor based operations overlap, whichever placed the descriptor object first will be resolved first. This is because if another operation sees the descriptor, it will complete it, otherwise it’s own operation will be completed first.

# **Usage**

Using Tervel requires a main thread calling an initialization function for the library and each thread that uses Tervel algorithms to call an attachment function. Memory reclamation is handled when calling the destructor of the object returned from the initialization or attachment function. The only thing the user must add in addition to the framework for their own algorithm to work is to implement an abstract class for the descriptor object.

# **Dependencies**

Tervel depends on g++ 4.8.0 or greater and gflags.

## RSTM

# **Overview**

The Rochester Synchronization Group developed the Rochester Software Transactional Memory (RSTM) system, a C++ package that contains thirteen different STM library implementations. It additionally provides a smart-pointer based API that allows consistent, high-performance, safe, and relatively transparent access to STM. It is advertised as a research prototype, however it is currently in it’s seventh release and has been successfully tested on a variety of benchmarks and applications.

# **Usage**

Everything in the RSTM API is provided in the stm namespace (except for the BEGIN\_TRANSACTION and END\_TRANSACTION macros). init must be called in each thread before it engages in any transactions. The BEGIN\_TRANSACTION and END\_TRANSACTION macros are used to mark the beginning and end of a transaction, respectively.

All transactionally protected classes are expected to inherit publicly from the Object<T> template. Fields are generated using macros provided by the library. There are macros for generating fields of any valid type, arrays and 2D arrays of said types as well.

Smart pointers are the primary method by which client code interacts with transactional objects. There are four possible states corresponding to the different transactional states that a shared object can be in: [11]

1. Shared - A shared object is one that has not yet been touched in a transaction. An example of a shared object is the "next" pointer of a linked list node, before it is read or written.
2. Read Only - A read only object represents an object that has been opened for reading in the current transaction. A read only object roughly corresponds to a const object.
3. Writable - A writable object is one that has been opened for writing by the current transaction. Both const and non-const members can be called on a writable object. No changes made to a writable object will be visible to other threads until the transaction commits.
4. Privatized - A privatized object is a nominally shared object that will only be accessed (read or written) by one thread at a time. Program logic is responsible for ensuring this invariant. Privatized objects may be safely read or written inside or outside of transactions.

Each state corresponds to the four types of smart pointers in RSTM, respectively:

1. sh\_ptr<T>
2. rd\_ptr<T>
3. wr\_ptr<T>
4. un\_ptr<T>

Different implementations of STM typically require reads (or possibly write) be validated after reading (or writing) in order to disallow inconsistent versions of data to exist. RSTM uses a Validator object to do this post validation. The actual validation is handled in the read accessor created by the macros used to generate fields. This validation happens internally and the Validator object is never exposed in client code.

The following code example initializes a rd\_ptr<Node> from an sh\_ptr<Node> (sentinel). Then, in order to read the next pointer, the code uses the get\_next(...) accessor, passing in the validator as returned from the smart pointer.

rd\_ptr<Node> prev(sentinel);

rd\_ptr<Node> curr(prev->get\_next(pref));

This pattern of calling a getter through a smart pointer and passing the validator associated with that smart pointer is used everywhere. [11]

RSTM also supports explicit privatization via the un\_ptr<T>. Privatization safety is provided by the transactional fence() library routine. (see **Criticisms of RSTM**)

## Boost

# **Overview**

Although, strictly speaking, Boost does not contain an STM library, there has been an attempt to include an STM package to the Boost library, which is of most interest to us due to the similarities in this attempt to our project.

Boost is a peer-reviewed portable C++ source libraries intended to be widely useful and usable across a broad spectrum of applications. Although not strictly a concurrent library, it does contain a number of useful algorithms and containers for parallelism, including:

* Atomic – atomic data types and operations
* Coroutine2 – templates for generalized subroutines
* Interprocess – simplified use of common interprocess communication and synchronization mechanisms
* Lockfree – Lockfree data structures
* MPI – message passing in high-performance parallel applications
* Thread – managing multiple threads and execution with shared data

Some of these libraries have already been included in C++ 11, however these Boost libraries are still useful for compatibility with older compilers.

# **Boost.STM**

Of interest to our project is a submission under the Boost wiki **Libraries Under Construction**. [12] Here there is a Boost.STM project developed by Justin E. Gottchlich and Vincente J. Botet Escribá. [13, 14] Although it seems to have been abandoned (last upload was September 19, 2009), it might be useful and informative if we reviewed the remnants of their project as we develop our own library for submission to Boost.

Aptly named Toward Boost STM (TBoost.STM), the project is a C++ lock-based software transactional memory library. TBoost.STM only uses native language semantics while implementing the least intrusive, most type-safe object oriented solution possible.

The key features that TBoost.STM provides are:

* Optimistic concurrency
* ACI transactions
  + Atomic: all operations execute or none do
  + Consistent: only legal memory states
  + Isolated: other txes cannot see until committed.
* Language-like atomic transaction macro blocks
* Closed, flattened composable transactions
* Direct and deferred updating run-time policies
* Validation and invalidation conflict detection policies
* Lock-aware transactions
* Programmable content management, enabling developers to specify forward progress mechanisms
* Isolated and irrevocable transactions for transactions that must commit (ie. I/O transactions)

# **Usage**

TBoost.STM is advertised to be in alpha release stage, with a disclaimer that although internal testing has proven the library to be fairly stable, the library may exhibit some stability issues. It is not a header only library and requires compilation before use.

Usage of the library requires adherence to the ACI principles. Observe the following example:

#include <boost/stm.hpp>

Boost::stm::tx::object<int> counter(0);

int increment() {

BOOST\_STM\_TRANSACTION {

return counter++;

} BOOST\_STM\_TRANSACTION

}

(A) Both the write on the counter and the read operations function atomically or neither operations are performed

(C) The transaction begins and ends in legal memory states, meaning counter is guaranteed to be read correctly, preventing thread data races from causing inconsistent results

(I) The intermediate state of the incremented counter is isolated until the transaction commits

These three attributes fulfill TBoost.STM’s conformance to the ACI principles.

# **Dependencies**

TBoost.STM depends on the following libraries:

* pthreads
* Boost.DynamicBitset
* Boost.Thread
* Boost.Pool
* Boost.BloomFilters
* Boost.Chrono
* Boost.containers
* Boost.InterThreads
* Boost.Move
* Boost.Synchro

# Languages

Some languages provide built in support not only for threads and concurrent data structures, but for transactional memory itself. The interesting thing about both these languages is that they are mostly functional.

There are several reasons why STM is more common in functional programming languages, one being that STM fits well with immutability. Eliminating side effects, ie., changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, alleviating some of the burden on the developer. Also, since STM is practically a research topic, programming language researches tend to prefer functional languages (a research topic in and of themselves). It also makes it easier to create “proofs” about program behavior.

## Haskell

# **Overview**

Haskell is a purely functional programming language, featuring a type system with type inference and lazy evaluation. It is widely used in academia and industry.

The STM library, as featured in *Composable Memory Transactions,* is part of the Haskell Platform.

# **Usage**

Users execute a block of actions as a transaction using the atomically combinator. Once the block is entered, other threads cannot see any modifications made until exit, nor can the thread see any changes made by other threads. These two properties mean that the execution is isolated.

Upon exiting a transaction, if no other thread concurrently modified the same data, all of the modifications will simultaneously become visible to other threads. Otherwise, the modification is discarded without being performed and the block of actions is automatically restarted. This all-or-nothing nature of an atomically block is referred to as *atomic*, hence the name of the combinator. [15]

In order to provide atomic, isolated transactions, it is critical that code does not deliberately or accidentally escape from an atomically block. Haskell’s type system enforces this via the STM monad.

ghci> :type atomically

Atomically :: STM a -> IO a

The atomically block takes an action in the STM monad, executes it, and makes its result available via the IO monad. This is the monad in which all transactional code executes.

## Clojure

# **Overview**

Clojure is a dialect of the Lisp programming language with emphasis on functional programming. It runs on the Java virtual machine and the Common Language Runtime. [18] The language encourages immutability, making it suitable for STM.

Clojure addresses the issue of state and concurrency by providing immutable data structures and language-level semantics for safe concurrency through managed references to state facilitated by software transaction memory.

The language also supports other systems for sharing state between threads, including the agent system, which supports sharing changing state between threads in an *asynchronous* and *independent* manner, and the atoms system, which supports sharing changing state between threads in a *synchronous* and *independent* manner. In all cases, Clojure does not replace the Java thread system, rather it works with it.

# **Usage**

Clojure exposes its implementation of STM via dosync, ref, ref-set, etal. It supports sharing changing state between threads in a synchronous and coordinated manner.

refs are mutable references to objects; while vars ensure safe use of mutable storage locations via thread isolation, transactional references (Refs) ensure safe *shared* use of mutable storage locations via STM. Refs are bound to a single storage location for their lifetime, and only allow mutation of that location to occur within a transaction.

Clojures implementation of STM opts for an approach based on Multiversion concurrency control (MVCC). MVCC maintains multiple (logical) versions of data referenced in a transaction. Threads operate on a snapshot of what the data looked like at the start of a transaction while in the scope of the transaction.

In practice, this means: [17]

1. All reads of Refs will see a consistent snapshot of the 'Ref world' as of the starting point of the transaction (its 'read point'). The transaction will see any changes it has made. This is called the in-transaction-value.
2. All changes made to Refs during a transaction (via ref-set, alter or commute) will appear to occur at a single point in the 'Ref world' timeline (its 'write point').
3. No changes will have been made by any other transactions to any refs that have been ref-set / altered / ensured by this transaction.
4. Changes *may have* been made by other transactions to any Refs that have been commuted by this transaction. That should be okay since the function applied by commute should be commutative.
5. Readers and commuters will never block writers, commuters, or other readers.
6. Writers will never block commuters, or readers.
7. I/O and other activities with side-effects should be avoided in transactions, since transactions will be retried. The io! macro can be used to prevent the use of an impure function in a transaction.
8. If a constraint on the validity of a value of a Ref that is being changed depends upon the simultaneous value of a Ref that is *not being changed*, that second Ref can be protected from modification by calling ensure. Refs 'ensured' this way will be protected (item #3), but don’t change the world (item #2).
9. The Clojure MVCC STM is designed to work with the persistent collections, and it is strongly recommended that you use the Clojure collections as the values of your Refs. Since all work done in an STM transaction is speculative, it is imperative that there be a low cost to making copies and modifications. Persistent collections have free copies (just use the original, it can’t be changed), and 'modifications' share structure efficiently. In any case:
10. The values placed in Refs *must be, or be considered, immutable*!! Otherwise, Clojure can’t help you.

# Criticisms of STM

While there has been much research in the field of parallelism, especially in regrards to STM, there has been some criticism of the methodology in general and interest in the subject has seemed of died down since 2010. [7, 11] Beyond that, there has been few practical applications of STM in the industry due to the complexity of implementing it, the overhead cost in adding it to an application, and frankly, the relatively marginal benefits of doing so.

For a time, Microsoft had begun development on software transactional memory for the .NET Framework (STM.NET). This was their attempt to provide the advantages of STM in the convenience and familiarity of the .NET framework. It also provided additional functionality, including:

* Annotations allowing the developer to designate:
  + Methods must either run in, not run in, or may run in a transaction
  + Fields must be accessed within a transaction
  + Suppress the transaction for this method
  + Redirect transacted calls to another method
* Static and dynamic checking of annotations
* Lock Interoperability

In 2010, Microsoft announced the discontinuation of the STM.NET experiment. [7] Joe Duffy, Microsoft’s best known researcher on parallel and concurrent programming, cited four reasons why he becomes disillusioned with STM in his *Brief Retrospective on Transactional Memory*. [9] These four points culminate the best criticism on software transactional memory.

First, is the issue of I/O. So far, STM has looked at operations consisting of pure memory reads and writes. This is only covers a portion of the majority of software written for applications today. What about reading from a file or writing to a console? Parallelism is is largely used in network communications as well; what about accessing data across a network?

Work-arounds are available, commonly, on-commit and on-rollback actions, to perform or undo the logical actions performed, respectively. However, this required extra overhead, as well as it’s own challenges, further adding to the development cost. Ultimately, it came do to the fact that not all operations are inherently transactional, and trying to adapt a transactional model to these kinds of operations led to a square-peg in a round-hole scenario. For instance, how do you undo a write to the console?

The second issue Duffy points out is the question of weak or strong atomicity. What happens if code accesses the same memory locations from inside and outside a transaction? Up until now, we’ve operated under the assumption that the memory we are operating off of is only accessed via a transaction. But if some location were to be accessed non-transactionaly whilst being operated on by a transaction, the transaction would no longer be safely managed and no guarantees could be made about it.

While there are hardware options being researched that support transactional memory, this leads us way from the territory of *Software* Transactional Memory. Furthermore, we do not want to have the integrity of an application depend on the hardware implementation it is run on. This issue comes down to either strictly requiring transactional objects to only be read and written transactionaly or else trust that the developer follow recommended practices.

The third issue with STM is privitization. Take for example the following code:

bool itIsOwned = false;

MyObj x = new MyObj();

…

atomic { // Tx0atomic { // Tx1

// Claim the state for my useif (!itIsOwned)

itIsOwned = true;x.field += 42;

}}

int z = x.field;

…

In Duffy’s own words:

The Tx0 transaction changes itIsOwned to true, and then commits. After it has committed, it proceeds to using whatever state was claimed (in this case an object referred to by variable x) outside of the purview of TM. Meanwhile, another transaction Tx1 has optimistically read itIsOwned as false, and has gone ahead to use x. An update in-place system will allow that transaction to freely change the state of x. Of course, it will roll back here, because isItOwned changed to true. But by then it is too late: the other thread using x outside of a transaction will see constantly changing state – torn reads even – and who knows what will happen from there. A known flaw in any weakly atomic, update in-place TM.

This problem had serious implications and led to further research into the matter. The intern who discovered this issue even went on to publish a paper proposing a solution to the problem. [10]

Finally, although not necessarily a fault of STM, there has yet to surface the “killer STM app”. Although the industry as a whole is still searching for a killer concurrency app, it has become increasingly likely that the killer concurrency apps that will be produced in the upcoming years will not even need transactional memory.

In conclusion, it ended up being rather difficult to lower the overhead of STM to the point where it would be generally appealing to developers. In addition, significantly better results need to be accomplished in order to make STM more compelling. In a research effort into STM, it was found that software optimizations managed to accelerate STM performance by 2%-15%. However, these gains were marginal in comparison to the overhead observed in implementing STM. [8]

Further research is recommended into the elimination of dynamically unnecessary read and write barriers. However, given that the usage of STM hinges upon its simplicity and productivity benefits, it’s unlikely any STM library will see much usage so long as any proposed solution to performance problems require extra work by the developer.

The transactional memory model itself, whether implemented in hardware or software, introduces complexities into software applications that tend to limit the expected gains in productivity, thereby making it less likely to see broad adoption of transactional programming outside of narrow applications.

**Design Summary**

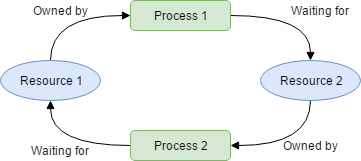
**Plan of Action**

**Related Work**

**Lock-free Transactions without Rollbacks**

**Blocking methods**

The concept behind lock-free concurrent data structures is to allow processors with multiple cores to run operations and instructions without the necessary bottlenecking and performance hit that comes with locks and blocking. Locks have been around since the dawn of concurrency, but they are often too complex for the uninformed programmer to implement properly. There could be the issue of priority inversion, in which lower priority threads have the key to a lock on an object, which causes higher priority threads to wait for that object or require the lower priority thread to be preempted; deadlocks can occur when two threads attempt to acquire the lock of an object in different orders. An example of deadlocks is shown below:

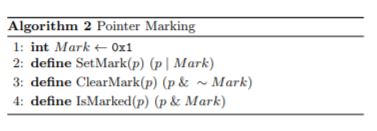


The example above shows what happens during a deadlock: Process 1 owns a resource, such as a Lock on a Node object in a linked list. In for Process 1 to proceed, it must acquire a resource, such as a lock for another Node in that same list, which is owned by Process 2. But, for Process 2 to unlock the Lock Process 1 needs, it needs Process 1 to finish and unlock its Lock. Because of this, Process 1 and Process 2 will never finish, and thus neither one will complete their desired instructions. Because of this complication, we must move on to non-blocking methods.

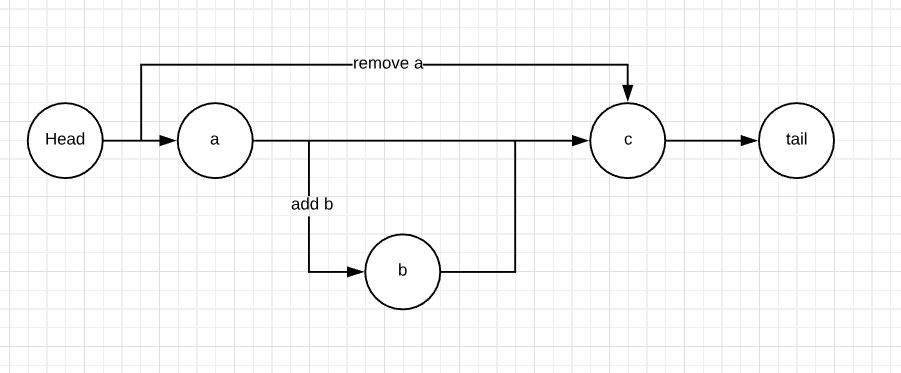
**Non-Blocking Methods**

**Single Operation Methods**

An example of using non-blocking methods on a data structure with one operation per thread can be observed inside of a linked list; the use of Non-blocking synchronization on a linked list utilizes methods such as compare-and-set in order to forego using locks. Non-blocking synchronization is best implemented through markabable objects, such as instances of the AtomicMarkableReference<T> object in *Java* or by bit-masking the final bit of an address of an object *C++*. *C++* allows for all addresses to be created with their final 2 bits always containing 0; this is done by using hex codes 0, 4, and C as the final hex digit in an address. Dr. Tim Harris implemented this non-blocking linked list by using the following bit-masking methods:



What is observed in this picture is that there is a global integer “Mark” that is set to 1. Because of the property that *C++* has of its virtual addresses containing 0’s in the last 2 bits, we can utilize bitwise operations to mark, unmark, and see if a node is marked based on its next pointer. This means that by utilizing bit masking or markable references, you can check to see if a Node in a List has been changed by one thread, such that another thread does not interact with that Node in the list in any observable way. An example of this would happen in the following execution of statements on a linked list pictured below



Say that Thread A desires to remove Node(a), while concurrently Thread B wants to add Node(b). If Thread A executes its instruction first, Node(a)’s next pointer should be atomically marked using the SetMart method, in that its final bit will be set to 1. This, however, only logically removes the Node from the list. It still exists in the list as a physical Node with memory allocated. What Thread B will do, then, during its traversal of the list, is notice that Node(a) has been logically removed by accessing the isMarked method on Node(a). Because that method will return true, Thread B will physically remove Node(a), which will prompt Head’s next pointer to point towards Node(c). After the execution of Thread B removing Node(a), Thread B knows that the list is safe from manipulation and can correctly place Node(b) inside of the list at the correct location. Node(b)’s next field will obviously not be marked, and can be safely traversed by other threads.

Implementing this application of marking a Node’s next pointer allows a thread to safely traverse the list without traversing over nodes that have been marked and believing them to be safe to manipulate. However, this implementation of non-blocking synchronization is only applicable when a thread is running a single operation on a structure. The obvious question that stems after implementing this methodology is: what happens if there are multiple instructions that a thread desires to execute? The non-blocking version of that linked list described above would have to terminate the operation that a thread was working on, go back to main, instantiate another thread to run the next operation, and rinse and repeat. There needs to be a better way to implement concurrency without the need for advanced knowledge of how to properly use locks, as well as utilizing threads and their ability to conduct multiple operations in one pass. That solution is known as a transaction.

**Atomic Transactions and Multiple Operations**

Atomic transactions can be described as a thread being able to execute either a whole set of read and writes seemingly at once without any interrupts; atomic operations get their name from the idea that an atom is the lowest unit of measure, in that you cannot divide it into subdivisions. This notion of subdivisions is described by the fact that an atomic operation either completely succeeds, or it does not succeed at all. There are atomic abstractions that can be implemented with hardware as well as software. We already covered the software version by using locks. The hardware implementation of abstraction exists in cache coherency, in that whenever a write operation on one thread occurs, that change is propagated throughout the rest of the threading in a timely manner. All of this is to say that, although atomic instructions do not happen actually instantaneously, there is the *appearance* that they do. An example of this phenomenon is when creating and modifying a long integer, such as:

foo = 65465498L;[[1]](#endnote-1)

This example shows a long integer of size 64-bit being and initialized. A multicore processor can take advantage of its threads by splitting the assignment into two sections: one thread will assign the most significant 32 bits of the number; the second thread will assign the least significant 32 bits of the number. This operation could create issues, though, as other threads or members may try to access the data as it is being assigned. Take, for instance, a situation in which a thread *A* is currently writing the most significant bits to the integer, thread *C* will write the least significant bits of the integer, and another

thread *B* attempts to access the data concurrently with thread *A*. In that instance, thread *B* is getting the version of the integer that has the format:

(new)(old),

where the new tag represents the newly assigned most significant bits and the old tag represents the previous least significant bits. This creates a dependency and correctness issue in that thread *B* is reading a value that is not what should be read. Thus, the system must make the changes to *foo* atomically so that *B* is getting the correct data; however, we cannot do this through locks, since we are attempting to get rid of locks. Thus, we must provide machine driven solutions to atomic instructions.

**Restrictions on Transactions**

The previous example shows one of the two biggest restrictions on transactional concurrent programming: to effectively buffer write operations in such a way that operations outside of the operation scope cannot “see” the modifications, yet complete tasks in a thread on a consistent basis. Because a thread might need to wait on the execution of instructions from other threads in order for it to access the right data, like thread *B* wanting to access *foo* with the correct data, a thread can complete the instructions of another thread without “seeing” what that other thread is changing. Take, for example, a thread A that is waiting on a thread B, as defined below:

Thread A Thread B

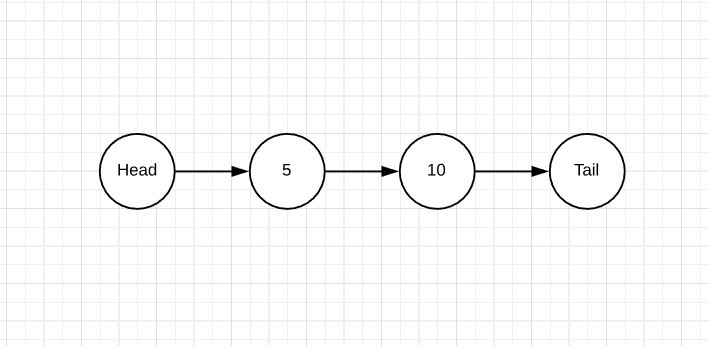
1. Add(a, b) Sub(c, b)
2. Add(b, b)

Thread A is attempting to add the value “b” to the value “a”; however, Thread B is in its execution cycle first. Thus, B is making manipulations to “c” and then “b” on instructions 1 and 2. If Thread B is only on instruction 1 by the time Thread A attempts to access “b”, what Thread A can do is execute Thread B’s instruction 2 by adding “b” to “b”. In doing so, Thread A can return to its execution cycle and manipulate “a” as it was originally intending. Thread A has no knowledge of the changes to shared memory that were done by Thread B’s instructions, it only knows that it was operating on data to drive it to completion, thus preserving the property of isolation. Write operations must be done in a fast and manageable manner so that there are not dependencies among threads as they individually execute their own instructions in their individual sandboxes. Making write operations conduct themselves in a more timely manner also makes it easier for the memory management unit to propagate data out to all of the available caches that threads are pulling shared data from.

The second great contention of transactions is the ability to conduct rollbacks on operations that must be aborted due to incorrect manipulations of data. Say and operation was to fail on a transaction, such as a segmentation fault. If that process fails, then there was an instruction in the transaction that did not allow it to go to completion. If the transaction did not go to completion, there must be some type of way to exit the sandbox, call an exception, and state that the transaction was not finalized, and thus data may be corrupted and incorrect. This poses a problem as other threads attempt to access data that may have been changed by the aborted thread, and these exceptions must be dealt with. Those two features of transactions have different handling methodologies, each of those with their own strengths and weaknesses.

**Another Example**

Although the example of assigning a double word a new value is a good representation of how transactions work, they do not show the whole picture of what a transaction contains. A transaction is a set of serializable instructions, in that those instructions must be conducted sequentially amongst a single thread. In the example of assigning a long integer, there is only one executable instruction per transaction, which defeats the purpose of calling it a transaction at all; we could simply use a lock free data structure that can execute one instruction at a time. A better representation of a list of transactions with multiple instructions per transaction would be operations on a linked list, like what was described above for non-blocking synchronization of a single operation. Take, for example, a linked list with a head node, two non-terminal nodes, and a tail node. A depiction of this example is shown below.



Now assume the two transaction T1 and T2 take place:

T1 T2

Insert(7) Insert(15)

Delete(5) Find(5)

Because transactions are happening simultaneously, operations on the shared linked list are happening simultaneously. Thus, the big question of transactional synchronization is represented in this example: what happens if Delete(5) from T1 is executed at the same time Find(5) from T2 is executed? The next three methodologies of lock-free transactions attempt to solve this problem, each in different ways.

**Software Transactional Memory**

Software transactional memory (STM) is one methodology that allows for non-blocking transactions to occur. STM is a form of atomicity that is centered around the idea of isolating the intermediary transactions that are taking place at any given time in shared memory.[[2]](#endnote-2) In essence, you are creating a sandbox for a thread to play in and make modifications to the data, while also allowing for no other threads to see those modifications; this statement, however, only holds for as long as the thread that is isolated does not exit from the block of execution. Upon exiting the block of execution, one of two things must happen: if no other thread has modified the same data in a different sandbox, then the data is pushed and made visible to all other threads. Per the example in the previous section, if Node(7) is not created in a thread other than T1, you may push Node(7) to shared memory using the Insert(7) call. However, if another thread *has* modified the same shared data, then we must discard the changes that we have made and start the process over. This idea of committing or aborting a transaction is based off what is known as *read/write conflicts*.[[3]](#endnote-3) A conflict arises when one object/data’s read or write set intersects another object’s write set. In other words, if one thread either writes or reads a piece of data as another thread is writing to that same piece of data, it could be corrupted or inaccurate. In those instances, simply get rid of the state of one of the sandboxed pieces of data and start the process over, while committing the other thread’s data. Using that same running example, because T1 modified the list by deleting Node(5), you cannot assume that T2 can accurately depict the list it started its execution with at the end, so you must discard all operations done by T2 in its sandbox.

T1 T2 (discarded)

Insert(7) Pushed Insert(15) X Not pushed to shared memory

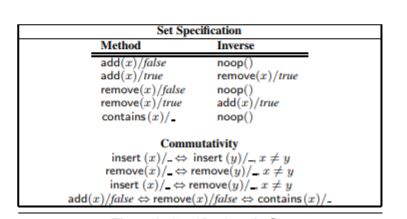
Delete(5) Pushed Find(5) X Not pushed to shared memory

STM obviously works very well from the perspective of the first contention of non-blocking transactions, in that STM ensures that buffers are created to ensure that threads are not affected by the manipulations of data by another thread. Threads will certainly run to completion of their instructions, including writes, no matter how long the operations take. No other threads are supposed to know what is happening inside of another thread, so there should be no dependencies between threads that require assistance across threads. However, this is a costly implementation of transactions due to the necessity for one thread’s data to be redacted and started over in the case of a contention; STM requires each thread to have its own sandbox, and no other thread can play in that sandbox. Given this, there is a lot of time wasted by a thread working to accomplish its instructions without the aid of another thread. Also, because the process must be started over again if there is a read/write contention, and there is a non-zero chance that another thread may create contention for data inside of that process, the additional computation that is required for this methodology is quite cumbersome. Thus, we must search for better and more efficient solutions.

**Transactional Boosting**

Because STM on its own is not efficient enough to handle large amounts of contention, transactional boosting was created. Transactional boosting is an extension of STM that focuses on translating highly-concurrent linearizable objects into highly-concurrent transactional objects. Linearizable objects can be described as objects that meet three conditional factors: all of the methods must be atomic, and by this there is a before and after state of that operation that is distinguishable by the overall state of the data structure; there must be only one pending operation from any thread; and finally, the operations cannot be “lazy,” or, in other words, queued to be done at a later time. Linearizable objects also must maintain the characteristics of holding an abstract state and a concrete state. By utilizing these states, processors can identify the precondition and postcondition of a method by comparing the abstract states before and after a method call, which satisfies the second requirement of linearizable objects in their methods being atomic. Transactional objects, on the other hand, have the requirements that you can backtrack through the conducted operations logically without having to scrap the acquired data entirely.

This idea of transactional objects is heavily implemented through transactional boosting’s use of object semantics, in that boosting needs to know the abstract state (such as a set or ordered list) of the object it is modifying, as well as the methods that affect that state (such as an add() or remove()). Another characteristic of transactional boosting is that for all of its member methods, those methods must have inverse operations; this property ensures that the transactions can be regulated inside of “black boxes,” such that transactions maintain their ability to abort any operations that were performed at the time of execution through backtracking. Transactional boosting handles the requirement of backtracking by, in each instance of calling a method, the inverse method of that operation is placed into a stack call. Should there need to be conflict resolution on the thread that is being manipulated, simply go through the stack and invert the manipulations done to the data structure through that thread call. If there are no conflict resolutions to perform, push the data to memory and free the stack of its contents. In the example from before, should T2 have to abort its operations after T1 deletes Node(5), it can simply call the inverse operations of Insert(15) and Find(5). It is imperative to the operations of transactional boosting that efficient inverses exist for the operations that are conducted when manipulating objects. A method that would violate this property, for example, is adding a node with a randomized member key that is generated in the method call, and then attempting to remove that node from the list after a contention recognition. Because there is no way to retrace the hashed value of the key, there is no way to remove the node that contained the initial key value, and thus the method violates the ability to allow for backtracking during an abort call. A description of the methods that exist for Set objects and their inverses is shown below:



The advantages of using boosted objects is obviously that there is no need to abort the entire operation that a contested thread has completed, simply pop the stack until the moment of contention and resume operations from there. Using the running example, T2 would simply have to call the inverse of Find(5), which, in this case, there is no inverse, and thus we are simply left with the inserted Node(15) from T2’s first instruction.

T1 T2

Insert(7) Pushed Insert(15) Pushed

Delete(5) Pushed Find(5) X Conduct inverse operation

This methodology, though, requires that additional space be created for the call stack of inverse operations; if there are a significant amount of inverse operations on the call stack, the space needed to store those operations is additive to the overall space complexity of the program.

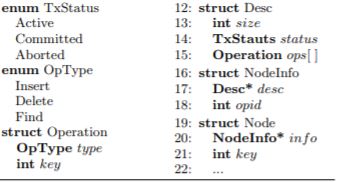
**Lock-free Transactional Transformation**

The two methodologies described above are good implementations to perform non-blocking synchronization. Those methods, though, provide issues in their design: there is additive overhead due to additional synchronization that takes place when comparing if there were discrepancies in terms of threads manipulating the same data concurrently. If there were overlaps, the need to conduct rollbacks or restart those processes makes the concurrent processes much less effective at their intended goal; there could be the need to retrace the call stack of inverse operations in boosting, or the need to discard all of the operations of a thread entirely in STM. Based off of this information, Dr. Damian Dechev and his team of graduate students at Area67[[4]](#endnote-4) implemented a new methodology for conducting non-blocking transactional processing.[[5]](#endnote-5)

This new methodology of concurrent programming, named *lock-free transactional transformation*, or LFTT, utilizes the same functionality of transactional boosting in that it saves the semantic representation of an object in order to reduce the number of possible false triggers and necessary aborts. This data is encapsulated in, what Dr. Dechev’s team calls, a “transaction descriptor,” which contains the operations, operands, and transaction status of the objects that are being manipulated. This implementation of concurrency is highly applicable amongst linked data structures, in which nodes are created to store data and are linked by reference pointers to each other; examples of these are linked and skip lists. The abstract state that exists inside of linked data structures is the set of integer keys that are created through hash algorithms, in which they are immutable and final upon creation. The concrete state of the data structures is the actual list itself; what nodes are accessible from the head node?

**How does LFTT conduct synchronization?**

Going back to the ideas of the contentions of concurrent programming (buffers and rollbacks), Dr. Dechev’s team innovated unique ways to handle those contentions. To handle the efficient buffering of write operations, the team has implemented a structure called the transaction descriptor that contains the instructions and arguments for specific transactions that can be performed. Implementing transactional descriptors in this way means that the synchronization technique must be “pessimistic”, in that whenever a structure is accessed, such as a Node, it is immediately assigned to the transaction that called it. This causes further transactions that desire to access the same data to assist in the completion of the transaction that first called the data piece. The specific transaction descriptor is implemented as a reference member field in each Node of the linked structure. The structure design implementation can be seen on the next page:



struct Desc: the transaction descriptor described above. Holds the status of an operation and the type of operations available (as enumerated by OpType and TxStatus)

struct NodeInfo: holds a reference to a transaction descriptor and an integer opid that can be used to reference the OpType

struct Node: holds a reference to a NodeInfo struct and whatever other fields the Node of a list would desire to have as members, like a \*next pointer.

To provide some clarity on the above code, here is what can be accessed when using the fields in those structures:

* Given a node n, n.info.desc.ops[n.desc.opid] is the most recent operation that accessed it.
* Active nodes have the property that n.info.desc.status = Active

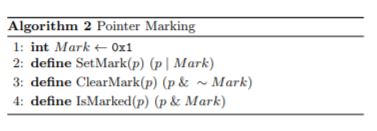
**Buffered write instructions**

Because each node has access to this structure, each node that is under a transaction will be added to a separate descriptor. Should a thread attempt to access a node, that thread will first read in the state of the transaction, as enumerated by the fields {Active, Committed, Aborted}. When a thread accesses the descriptor in order to conduct a transaction, it will only move forward in that transaction if the previous transaction on the acting Node has committed or aborted. If that is not the case, in other words, the other transaction’s status is Active, then that thread will assist the other transaction in its completion. This is supported by going through the other thread’s transaction descriptor and pulling operations from its ops array. By allowing threads to assist each other in their operations, write operations can be conducted at a much higher rate, thus furthering the process of concurrency. This ensures that the property of buffering write operations is maintained.

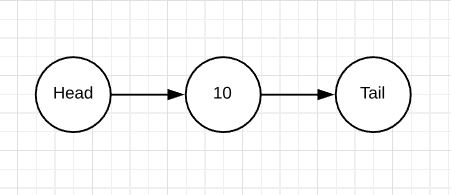
**Rollbacks**

In order to ensure that rollbacks are possible during an aborted state, the team has implemented “logical rollbacks,” in which the data structure, such as a linked list, can interpret the “logical status” of the node it is accessing. Previous implementations of logical status have been done by bit marking the abstract state, as to logically “remove” the object from the concrete state. For linked data structures, this means that there is no key associated with a Node. An example of this is bit masking a Node’s key member so that it no longer references a key in the table of accessible nodes (the keys that exist in the concrete nodes).

This type of bit masking can be described by the following algorithm:



Dr. Dechev’s team has extended this ideology of logically changing the state of a data structure by utilizing a node’s transaction status and its operation type. By way of interpreting the inverse representation of an instruction on a failed operation, one can interpret the abstract and concrete status of a Node in a linked structure. Using the example from above, assume T1 was successful in executing its instructions. Thus, the linked list now looks like this:



And the transaction descriptors now look like this:

T1 (Committed) T2 (Active)

Insert(7) Pushed Insert(15)

Delete(5) Pushed Find(5)

What will happen next is T2 will run through its execution and get to Find(5). Find(5) will go to that Node and see the transaction descriptor that just recently executed instructions on it, per < n.info.desc.ops[n.desc.opid] >. That operation will see that the descriptor for that node was Committed, and thus the operation Delete(5) was successful. Because of this, Find(5) can move forward smoothly, knowing that the altered linked list is not only logically different, but physically different. Find(5) will return false, due to the list not containing a Node(5), and thus the transaction descriptors will look like this:

T1 (Committed) T2 (Committed)

Insert(7) Pushed Insert(15) Pushed

Delete(5) Pushed Find(5) Pushed

That running example shows the effects of LFTT when an operation successfully pushes changes to shared data; but, what happens in a situation in which T1 aborts its operations, i.e. T1.Status = Aborted, and T2 attempts to access a Node that T1 was altering?

T1 (Aborted) T2 (?)

Insert(7) Pushed Insert(15)

Delete(5) Aborted Find(5)

In that instance, T1 attempted to delete Node(5), but that operation was not conducted. Thus, in the field n.info.desc.status, where n = Node(5), there is a status of Aborted. Now, when T2 attempts to access Node(5) through its Find(5) operation, it will access that same field, n.info.desc.status, and see that the status of that operation was Aborted. Thus, the operation Find(5) cannot proceed, and will thus have an aborted status, alongside its aborted transaction status.

T1 (Aborted) T2 (Aborted)

Insert(7) Not Pushed Insert(15) Not Pushed

Delete(5) Aborted Find(5) Aborted

The logical interpretation of those two transactions is that there was no Node that contained the abstract key of 5. Thus, because of the property of atomicity, the whole transaction T1 cannot be allowed to produce meaningful output. Because of this, even though the transaction could successfully push Node(7) to shared memory because it shared no dependency with the Nodes that caused the aborted status, Node(7) was not added in the concrete state and its logical interpretation is “not inserted”.

**Silo Database**

**Databases**

A database is a storage system where data is collected and stored in a structured fashion. The stored data is usually organized in such a way that access to the data is simplified based on certain requirements or search-criteria. Every database has a particular model that it follows, and the layout of the data is what typically defines the model. Databases are usually modeled to enhance efficiency.

Often, the term database is used to refer to the Database Management System, which is the software used by end users to perform manipulative and/or administrative tasks on the database itself.

Databases are typically modeled in many ways, but two large distinctions can be made when classifying databases. These two distinctions can be made by determining whether the database is a relational database or if it is a non-relational database. Relational databases have dominated the popularity race since the 1980’s and maintained this dominance for 20+ years. In the early 2000’s, the rise of the non-relational databases came about.

As far as a database’s physical attributes, they are typically stored on database servers, which are computers whose sole responsibility is to run the database management system and any of it’s related software. These database servers are usually multiprocessor computers and they typically have very generous memory capabilities and utilize RAID for stable storage.

RAID allows the processors to combine their components into one, (or more), units which allow for data redundancy and performance improvement. This data redundancy can allow for the checking of corrupted data, backing up lost/defected data, or even improving database response times when queries are ran against a database. When used for performance improvement, this approach can be considered as a form of database denormalization. This is typically avoided in design implementations, as database normalization is the more popular approach, being that this approach makes the best usage of storage.

**Transactions**

Data is stored into these databases with the use of transactions. Transactions are simply the term used to refer to a unit of work, which is performed within a database management system against a database. These transactions are independent of other transactions that might be taking place at the same time. In general terms, a transaction generally represents any change that is made to a database.

When transactions are performed, there are two main purposes that are targeted. One of the purposes is to allow for failure-recovery and to keep the database consistent even when errors/failures occur. The second purpose of transactions is to provide isolation between programs which may be accessing the database at the same time. Without isolation, two programs might access data at the same time, which can lead to errors in the data being retrieved or modified.

The make up of transactions in a database management system can include one, or multiple operations. This is similar to transactions done in a bank account. If a transfer were to be made from one bank account to another, a withdrawal with be made from one account, and a deposit will be made to another. These two operations are embedded into the one transaction that was executed.

**ACID Properties**

Each transaction in a database must adhere to the ACID (Atomicity, Consistency, Isolation, Durability) properties. What these properties ensure is that the transaction being performed is valid even in the event of errors/failures or any other issues that may arise. Each operation in a database transaction must satisfy the ACID properties, otherwise it is not a true transaction.

When ensuring the adherence of these properties, a transaction can ensure that it is compliant with the Atomicity property so long as it is treated as a single unit. Whether this unit successfully executes, or it fails completely, it must guarantee that it is a single unit in every situation, regardless of environment issues (errors, crashes, failures, etc.).

A transaction can consider itself to adhere to the Consistency property if the transaction transforms the database from one valid state to another valid state. These state changes must maintain valid data that complies with all of the defined rules. Consistency prevents databases from being corrupted by illegal transactions. Although Consistency prevents corruption, it does not guarantee that the transaction is correct.

A transaction which follows the Isolation property basically ensures that the transaction will leave the database in the same state if the transaction is ran simultaneously with other transactions or if it is ran alone in a sequential pattern. Being that many transactions can run at any given time, a transaction that follows the Isolation property guarantees that it can run concurrently with other transactions without collisions.

Our last element of the ACID properties is the property of Durability. This guarantees that once a transaction has been committed to a database, it will remain in the database regardless of any system issues that may present themselves. Durability ensures that during a power outage or system crash, the data that has been previously committed still exists in the database, and based on the database’s recovery features, there is a potential for the data to be recovered. Although recovery does not fall under the rules of Durability, the adherence to durability leads to a system’s ability to implement recovery features, which are vital in database implementations.

With our basic understandings of ACID, we can now look at what may happen if a transaction fails to correctly implement each of these properties correctly.

In the case of Atomicity, if a transaction fails to be atomic, the entire transaction will fail. If a transaction contains many operations, either each operation will occur, or each operation will fail. Successful operations cannot be separated and executed in a transaction where another operation failed because this will lead to the database being partially-modified, and this will lead to greater issues than it would to reject the entire transaction in full. This can be seen with our prior example of the bank accounts. Consider two bank accounts, A and B. If we attempted to execute a transaction that would transfer $100 from Account A and into Account B, we would perform a withdrawal from Account A. Assuming the withdrawal was successful, we try depositing the $100 into Account B, but an error occurs. Without atomicity, we will apply the changes to Account A, but Account B goes unchanged, leaving us with a partially implemented transaction. It would be far more beneficial if we were to cancel the entire transaction once we received a failure from the operation on Account B. In the case of a successful operation, we would not update any data in the database until each of the operations in the transaction were successfully executed. In the example above, once the $100 was successfully deposited into Account B, we would then complete the withdrawal from Account A (which we already know executed successfully).

At this point, we’ve achieved Atomicity, and now we check for Consistency. Consistency requires that all of the data meet the validation rules set out by any constraints that may have been implemented. See the example below:

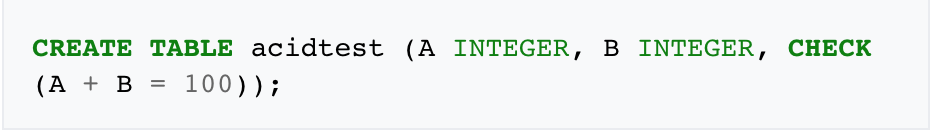


Figure 1

Here, we see that we are creating a table called “acidtest”, with two columns, A and B, and a validation rule. In this example, the requirement is that A + B = 100. Let’s consider a situation where a transaction attempts to subtract 10 from A without changing the value of B. Because consistency is checked after each transaction, our transaction knows that A + B = 100. If the transaction successfully subtracts 10 from A, we have a fully atomic transaction, but when checked for consistency, we will see that the transaction is not valid because the check for A + B is now equal to 90. This violates the rules set for the database, and consequently leads to a failed transaction. Due to this failure, any changes that were made will be cancelled and the state of the database will be returned to its pre-transaction state. Another example can be shown if the transaction were provided an incorrect data type as A or B. These two are requiring the INTEGER data type, so if any other data type were provided, the transaction would fail and prevent any changes from being made to the database. Consistency promotes the integrity of the data being manipulated in our database and guarantees our data is correct.

If our transaction is Atomic and Consistent, it must now be checked for Isolation. As you can recall, Isolation is the notion that the transaction will provide the same results whether executing simultaneously with other transactions, or alone. If we consider the same sample SQL statement in Figure 1, and we consider two transactions, Transaction 1 and Transaction 2, consider the situation where Transaction 1 would like to subtract 10 from A and add 10 to B, and Transaction 2 would like to do the inverse of Transaction 1, namely, add 10 to A and subtract 10 from B. With these two transactions, both performing two operations, we find ourselves with a total of 4 operations being executed. Not considering Isolation, if each operation in the two transactions are executed successfully, we find ourselves with a successful database state containing valid data. But consider a situation where our transactions operate in this fashion:

Transaction 1 subtracts 10 from A

Transaction 2 subtracts 10 from B

Transaction 2 adds 10 to A

Transaction 1 adds 10 to B

In a situation where Transaction 1 fails to add 10 to B (step 4), we will find ourselves in a situation where A’s value has been modified based off of Transaction 1’s modification on A. This leads to a write-write failure, and A cannot be restored to the value it had before Transaction 1 performed its first operation. This will lead to an invalid database state. On the contrary, if these transactions implemented the Isolation property, these two transactions would perform their operations in isolation of each other, preventing the situation where the same data is accessed.

Our last component of ACID is Durability, and the following example can show its importance to transactions and having a reliable database. Let’s revert back to our example of the bank accounts. Considering two bank accounts, Account 1 and Account 2, we withdraw $100 from Account 1 and deposit it into Account 2. After these two operations execute, the user receives a notification that the transaction was successful, but the changes are still queued on the disk buffer waiting to be committed. While this transaction is queued, if the power of the system goes down, this transaction will never be committed to the database, therefore, the changes don’t exist. This transaction was not durable which leads to a lack of data integrity, as the user is expecting a certain amount to be in one account, and instead, the system reflects different data values.

**Transactional Locking**

With our fully compliant transactions, we can take a slightly deeper look at how some of these characteristics are achieved. When transactions are implemented, there are many issues that may occur to prevent a successful transaction. In order to provide ACID capabilities, many databases implement locking functionalities. Locking, as the name states, is when a transaction leaves a mark on some data that it is accessing so that the database management system prevents other transactions from accessing this specific data, basically “locking” the data from access to other transactions. This lock is always acquired before processing the data so that the transaction is guaranteed to be processing the correct data, and is guaranteed to have the correct data throughout the entire transaction.

A downfall to traditional locking is that transactions can tend to be very dependent on locks, and usually require the use of a very large number of locks. This heavy use of locks results in substantial overhead and also leads to the blocking of other transactions. If we consider an example where Transaction 1 is reading a row of data that Transaction 2 would like to modify, Transaction 1 will have a lock on the data, and Transaction 2 will have to wait until Transaction 1 has finished with it’s entire transaction.

In order to circumvent the potential downfalls of locking, some databases may use the multi-version concurrency control technique, where the database provides all of the “read” transactions with the unmodified version of the data that is being modified by another transaction. This technique removes the need for locks because writing transactions are not effecting reading transactions, and vice versa.

**In-Memory Databases**

These traditional databases have been in use since the 1980’s and have continued to gain popularity throughout the years. In the 2000’s, in-memory databases (IMDBs) were introduced to promote quicker data access and better performance. Rather than storing data on a disk, in-memory databases store their information in the main memory (RAM). This change in data access is what allows for quicker data access and efficiency because disk access is slower than memory access, and this leads to the internal optimization algorithms to be much more simpler, resulting in fewer CPU instructions being executed.

IMDBs are typically beneficial to applications whose response time is very critical for success. IMDBs have gained popularity in the world of data analytics starting in the mid-2000s, and this was largely due to two factors, the first factor being multi-core processors that were able to address large memory and the second factor was less expensive RAM.

**Durability with In-Memory Databases**

Despite the many benefits of IMDBs, they do face an issue, which is the volatility of RAM. RAM is accessible when the system is on, so in the event of an unexpected power loss, any data stored on the RAM is lost. There are potential work-arounds to circumvent this issue, and non-volatile RAM technology is one of these work-arounds. With non-volatile RAM, in-memory databases will be able to run at full speed and maintain it’s data if a power loss were to occur.

This last notion of volatile RAM violates the durability property of the ACID property, but there are a few methods that can be used to help maintain this property. A few popular methods to do so are taking snapshots of files, also known as taking checkpoint images, and what this does is record the state of the database at any given moment of time. They system typically generates these snapshots periodically, and usually when the system does a controlled shutdown. Although this gives us a semi-reliable solution to our durability issue, this solution may lose recent changes that were made if a snapshot wasn’t taken before a system failure. To allow for full durability, snapshots can be implemented with Transaction logging, Non-Volatile DIMM, Non-Volatile RAM, or High availability. I’ll further elaborate on these methods in the following paragraphs.

The first supplemental method to fully enabling database durability is Transaction logging. This promotes durability because it keeps a journal file that is used to record any changes that are made to the database, and it facilitates the automatic recovery of an in-memory database. With a log of each transaction in a database, a system can easily track any changes, and in the event of a system failure, transactions can be repeated in order to recover the state of the system before it failed.

Another supplemental method that would work hand-in-hand with snapshot files would be Non-Volatile Dual In-line Memory Module (NVDIMM). The benefit of this method is a memory module that has a Dynamic Random Access Memory (DRAM) interface, which is often combined with NAND flash which is what provides it with it’s Non-Volatile data security. NVDIMMs are designed with supercapacitors instead of batteries, and this unique power source is what allow NVDIMMs to resume securely from the same state after a reboot.

Another method that can be considered to work hand-in-hand with the “snapshot files” method would be Non-Volatile Random Access Memory (NVRAM), and this typically functions in the form of static RAM that is backed up with batter power, or an electrically erasable programmable ROM. This type of storage is what allows the IMDB system to recover the data it’s last consistent state after reboot.

Our last supplemental method is High availability. This relies on database replication and uses an automatic failover to an identical standby database just incase the primary database fails. The IMDB is typically replicated in order to protect against a loss of data in the event of a complete system crash. This method is usually incorporated with other mechanisms listed above.

**SILO In-Memory Database**

There are a few implementations of IMDBs, SILO being one of them. SILO is considered to be a multicore IMDB that allows for speedy transactions to take place. This was a joint project between MIT and Harvard, and the main objective was to create an IMDB that would provide fully serializable transactions to ensure the Consistency property stayed intact. Another objective was to ensure the database’s durability and ensure that the database could recover from system failures/crashes. MIT and Harvard decided that multicores would provide the best value in implementing such a database due to the vast amounts of available memory on a multicore unit.

The beginnings of SILO came about in an attempt to build an IMDB on a multicore computer structure using parallel programming and an advanced B-Tree to provide a transactional database. This system was successful up to 16 threads, but had a drop in performance when performing with more than 16 threads. After running benchmarks, it was found that the use of a Global Transaction Identifier (TID) was the heart of the problem. TIDs were used in recovery stations to ensure the databases were recovered in the proper and valid state. Although very useful, the use of TIDs is not scalable. Researchers at MIT and Harvard found that the use of TIDs weren’t entirely necessary, and they started the development of the SILO database which would utilize a modified version of TIDs.

With SILO, the database is guaranteed near-linear scalability and the raw numbers reported by system are higher than those seen by existing state-of-the-art transactional systems.

SILO’s main structure is using a scalable and serializable transaction commit protocol. This means that shared memory contention occurs only when transactions conflict with each other.

SILO’s implementation of such a system was to use time-based epochs to avoid doing the serialization write for each transaction that occurred. To do this, each transaction that comes into the system will receive a sequence number and an epoch. The sequence numbers are a means of providing serializability during execution, and epochs are used for recoverability. The combination of this sequence number and the epoch serve as a similar structure to the TID which was mentioned earlier.

**SILO Structure**

SILO’s underlying structure resembles that of a Masstree. A masstree is basically a fast and concurrent B-tree like structure that is optimized for multicore performance. During the implementation of this structure, the researchers realized that Masstree’s only support non-serializable transactions that are single-key. This became an issue because real databases need to support transactions that affect multiple keys and occur in serial order. In order to circumvent this issue, SILO uses it’s own commit protocol, which provided the properties that were missing from the Masstree methodology.

**SILO Commit Protocol**

SILO uses a pre-commit execution phase that is similar to that of the Standard Optimistic Concurrency Control (OCC) protocol. In OCC, a transaction keeps track of the records that it reads and writes to in thread-local storage. Once the transaction is ready to commit, the OCC protocol verifies that none of the concurrent transactions in the write-set overlap with those of the read-set, and once this validation is complete, the transaction will commit it’s written records at one time.

In a situation where the validation fails, and the protocol observes overlapping members of the write-set when compared to the read-set, the entire transaction will fail. This protocol has many benefits for scalability. Being that OCC only writes to memory once it is ready to commit its writes, this short write period reduces contention with other transactions. Another helpful characteristic of OCC lies within the validation step. As a result of this validation step, read-set records do not need to be locked. Being that locks are not required, memory writes do not need to contend with locked memory reads.

Although a multitude of benefits, there are some drawbacks that have been observed with the OCC protocol. Previous OCC implementations are not free of scaling bottlenecks, which is a result of OCC tracking anti-dependencies (write-after-read conflicts). If we consider an example with two concurrent transactions, Transaction 1 and Transaction 2, and consider the case where Transaction 1 attempts to read a record from a database and Transaction 2 overwrites the value that Transaction 1 read, a typical serializable system should order Transaction 1 before Transaction 2, regardless of any crashes and recoveries from the logs. In order to achieve this ordering, many systems require that these transactions communicate with each other through shared memory where one transaction would post it’s read-set to shared memory. Another method is the use of a unique transaction identifier.

SILO, instead, uses a different approach that is able to provide serializability as well as avoid the use of shared memory writes during read transactions. In SILO’s commit protocol, the researchers decided to design it in a way that used memory fences that would produce results that were consistent with serial order in a scalable manner. Although serializability was solved, the researchers still had to overcome the issue of correct recovery. In this system, correct recovery is achieved by using a form of an epoch-based group commit. This method divides time into a series of short epochs. This allows the system to always agree with serial order without explicitly knowing the serial order. Simply put, if Transaction 1’s epoch is before Transaction 2’s epoch, they system knows that Transaction 1 precedes Transaction 2 in serial order. This allows SILO to provide the same guarantees as any serializable database without the extra scalability bottlenecks and the additional latency. Aside from serializability, these epochs can be used to provide database snapshots, which can help reduce aborts from read-only transactions.

Essentially, to read a record, the record’s TID is saved in a local read-set, then the value is used. For writing, the write is saved in a local write-set, and the system keeps the records unmodified.

To commit changes in SILO, there is a commit protocol which consists of 3 phases. During the first phase, all records in the write-set are locked while the current epoch is read. During phase 2, the records in the read set are all validated. In this phase, if there exists a record whose TID changed, or whose lock is already being held by another transaction, the current transaction will be aborted. The last phase of the commit protocol involves picking the TID and performing writes on it. In this phase, the epoch recorded in phase 1 will be used for the TID.

**SILO Potential Benefits with LFTT**

Although extremely speedy, SILO can be given even greater speed gains if it were able to use functionality that would disabled the need for locking on the write-set. SILO currently works in 3 phases, where the first phase locks the write-set while read operations are done, then the second phase validates that none of the records in the read-set are changed. The last phase simply performs the needed changes and commits them.

An issue that presents itself in this implementation of Silo is that when two transactions conflict (overlapping read/write sets), one of these two transactions will abort and restart. This causes a very large performance overhead and another issue exists where low-level memory conflicts cause excessive aborts.

A viable solution to the issue of conflicting transactions is Transactional Boosting. In this methodology, each node has a lock, and before performing an operation on a node, you acquire that node’s lock. If there is a failure when attempting to acquire the lock, then abort the transaction. The issue with this is that the system must physically roll back any of it’s completed operations, which is a very costly task. A solution that avoids excessive aborts and excessive rollbacks is described below.

Consider a situation where there was no need for locking to be done on the write-set in the SILO database. We would eliminate phase 1, but we would also have the luxury of no longer needing phase 2. Of course, these two phases would need to be replaced by some other functionality in order to ensure that the proper writes are made, but we can replace phase one and phase two with a mechanism called Lock-Free Transactional Transformation (LFTT). The LFTT methodology employs transaction descriptor objects in order to announce the transactions globally so that delayed threads can be helped. This methodology works great with quite a few linked data structures, such as linked lists, binary trees, and skip lists. LFTT encapsulates all operations, operands, and transaction status in a transaction descriptor, and this is shared amongst the other nodes that are accessed by the same transaction. LFTT also coordinates threads to help finish the remaining operations of delayed transactions based on their transaction descriptors. When a particular transaction fails, the correct state is recovered by reversely interpreting the logical status of a node. In this methodology, write operations become invisible to operations outside of the scope of the transaction until the transaction commits.

Typical Lock-free data structures have multiple threads that operate on the data structure concurrently. This notion of concurrent execution can appear as though the execution is occurring sequentially. Although not a bad thing, there does exist an issue with Transactional data structures where these transactions only focus on one operation at a time. Therefore, given a situation where there is a sequence of operations that need to be executed together in one transaction, this is unable to be done. The desired properties of such a structure would be atomicity and isolation. With these two properties, either the entire transaction will succeed, or the entire transaction will fail, as this system allows for no partial results while these multiple operations are performed. The isolation property will also allow the same notion of sequential execution, and any interleaved threads should not be visible to the user.

**LFTT Structure**

LFTT works heavily on the use of transaction statuses. There are three specific transaction statuses that are utilized in LFTT. An active transaction status means that the transaction has not completed yet. A committed transaction status means that the transaction completed successfully. Lastly, an aborted transaction means that the transaction failed completely.

The benefits of using Lock-free Transactional Transformation is that it detects conflicts without the use of locks, it eliminates the need for physical rollbacks when conflicts occur in acquiring a node’s lock, and it also has a thread-helping scheme which reduces aborts.

The thread-helping scheme that is implemented in the LFTT methodology is simply the use of descriptor objects. A descriptor in this methodology contains all of the information about any specific transaction. This information can include a list of operations performed, as well as the transaction status of the transaction itself (whether it is active, committed, or aborted).

These descriptors are also shared between all threads, and this is where the “helping” characteristic comes into play. When a transaction operates on a node, it makes that node point to the descriptor to check its status.

Regarding the elimination of physical rollback, the reading of the descriptor’s status is what enables us to prevent rollback. Every node will have a status that starts in the active status. The linearization point of a transaction is when you CompareAndSwap the descriptor’s status to a different state (Committed or Aborted). Any future transaction will use logical interpretation, which means it will determine the state of a node by observing the status of the descriptor that the node points to.

The thread-helping scheme can be elaborated in the following example. Assume we have two Descriptors, 1 and 2. Descriptor 1 would like to find(a) and find (d), meanwhile Descriptor 2 would like to insert(d) and delete(a). These two transactions will work simultaneously, so when 1 performs find(a), it’ll report success, just as 2 will report success when insert(d) is performed. When 1 goes to find(d), it will run into an issue where it’s accessing a descriptor in Active status, and instead of rolling back, 1 will help 2 execute the remainder of it’s operations, then resume it’s execution.

There does exist an issue with this methodology, which lies in the event of a circular dependency. If two transactions are accessing the same nodes in opposite order, this can cause a livelock. The researchers of LFTT came up with a solution to this, which is a Thread-local help stack. Any thread that helps another thread will be placed on this help stack, and if we get a duplicate entry on the help stack, we know we’ve come across a circular dependency, so we must abort one of the transactions.

There are strong arguments for LFTT and it’s benefits that it can bring to concurrent data structures. The first being that it is a lock-free methodology utilizing a helping scheme that ensures all threads make progress. Another benefit is that LFTT reduces unnecessary aborts. The conflicting threads help each other finish rather than aborting and starting over. A third benefit of LFTT is that this methodology saves time on rollback. Any aborted transaction does not need to physically rollback any of it’s completed operations because the transaction can logically interpret the existence of a node based on the transaction status.

**Linked List Example**

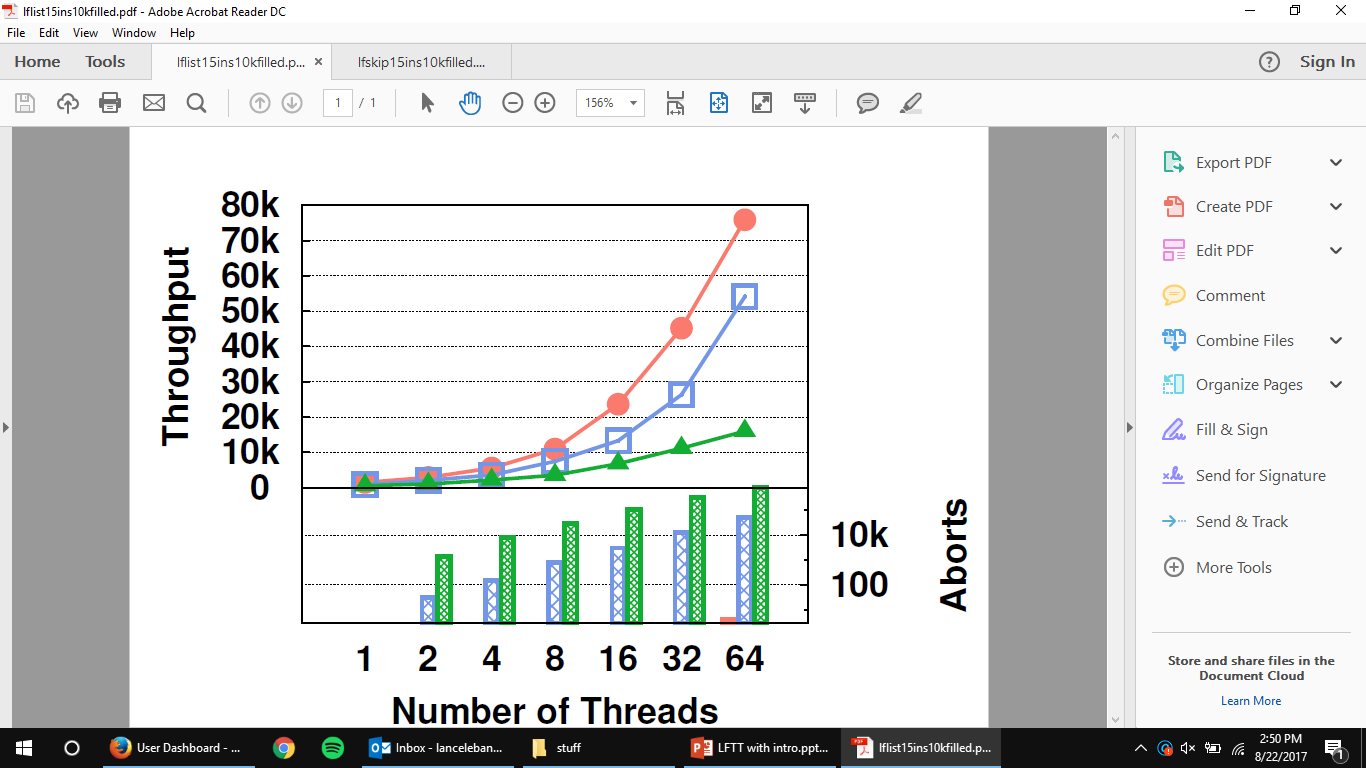


Figure 2

Figure 2 denotes the graph of the LFTT methodology (in orange) compared to the Boost methodology (in blue) and the STM methodology (green) when implemented on a linked list data structure. The graph shows the throughput in comparison to number of threads being ran, where throughput is measured in commits per second. We can also see the low level of aborts experienced in the LFTT implementation when compared to that of Boost and STM.

<https://dl.acm.org/citation.cfm?id=3209690>

<https://dl.acm.org/citation.cfm?doid=2935764.2935780>

<http://db.csail.mit.edu/pubs/silo.pdf>

<https://shekhargulati.com/2018/08/25/the-minimalistic-guide-to-acid-transactions/>

<https://www.codeproject.com/Articles/522039/A->

<http://hsqldb.org/doc/guide/sessions-chapt.html>

<https://www.c-sharpcorner.com/article/transaction-in-net/>

**Facilities and Equipment**

**Administrative Content**

* Milestone chart
  + We’ll have two versions of this
    - The one we made when we first got the project and met with Dr. Dechev
    - Another one that was updated and showed if we met our desired deadlines. For example, we didn’t meet the requirement for this past week (10/29) to refactor code. There should be a field in the table stating whether or not we met a deadline.
* Project summary and conclusions

1. https://stackoverflow.com/questions/15054086/what-does-atomic-mean-in-programming [↑](#endnote-ref-1)
2. http://book.realworldhaskell.org/read/software-transactional-memory.html [↑](#endnote-ref-2)
3. http://cs.yale.edu/homes/ejk/papers/boosting-ppopp08.pdf [↑](#endnote-ref-3)
4. http://area67.cs.ucf.edu/ [↑](#endnote-ref-4)
5. http://delivery.acm.org/10.1145/2940000/2935780/p325-zhang.pdf?ip=216.189.219.82&id=2935780&acc=CHORUS&key=4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E6D218144511F3437&\_\_acm\_\_=1539309860\_382100eb4c1db673d22f111afe34af10 [↑](#endnote-ref-5)