ABSTRACT
Software transactional memory (STM) is a programming paradigm that may be influential in the near future. We discuss the motivations for STM, its relationship with database concepts, and the state of current implementations. We close with an example of a concurrent data structure implemented in a current STM, TinySTM.

1. INTRODUCTION
The era for which the clock cycles of a CPU have been steadily doubled is finally over. Vendors have announced that they will start focusing on manufacturing multi-core processors as an alternative. On that account, now is the time for developers to consider parallel programming seriously more than ever to fully harness these multi-core chips.

However, people realized that it is very difficult to develop parallel programs for processors with increasing number of cores. Because application developers have to face with burdens such as synchronization tradeoffs, deadlock avoidance and races. In this environment, Transactional Memory (TM) has been proposed as a new parallel programming model that is easy and efficient for developing parallel software.

Briefly speaking, TM is a concurrency control paradigm that provides atomic and isolated execution for regions of code. TM is considered by many researchers to be one of the most promising solutions to address the problem of programming multi-core processors. Its most appealing feature is that most programmers only need to reason locally about shared data accesses, mark the code region to be executed transactionally, and let the underlying system ensure the correct concurrent execution. This model promises to provide the scalability of fine-grain locking, while avoiding common pitfalls of lock composition such as deadlock.

Unfortunately, the technology to enable the paradigm is still far from maturity, and we believe that the road ahead for STM is quite challenging. Lowering the overhead of STM to a point where it is generally appealing is a difficult task, and significantly better results have to be demonstrated. Some researchers are even skeptical insisting that the overall performance of TM is significantly worse at low levels of parallelism, which is likely to limit the adoption of this programming paradigm. Currently, a lot of research is still ongoing to resolve issues like this, and the progress is being made slowly but steadily.

2. TRANSACTIONS IN DATABASES
The problem of consistency is nothing new in the world of computers. In fact, it has been central to the entire solution in one particular area: databases. That is, the concept of transaction was firstly used in the Database research area. A database has to be updated atomically for the reason already stated: consistency of the data. It must not happen that one part of the update is performed while the rest is not. It also must not happen that two updates are interleaved so that in the end only parts of each modification are visible.

The solution in the database world is transactions. Database programmers explicitly declare which database operations belong to a transaction. The operations performed in the transaction can be done in an arbitrary order and do not actually take effect until the transaction is committed. If there are conflicts in the transaction, the transaction is rolled back and has to be restarted.

Like database transaction, TM has Atomicity, Consistency and Isolation (ACI) properties:
- Atomicity
- Consistency
- Isolation

General descriptions about ACID properties can be found easily on the internet. Interested readers can start looking into this excellent article [1].

3. TRANSACTIONAL MEMORY
The concept of the transaction is something that falls out of most programming tasks quite naturally. If all changes that are made as part of a transaction are made available atomically all at once, the order in which the changes are added to the transaction does not matter. The lack of a requirement to perform the operations in a particular order helps tremendously. All that needed is to remember to modify the data sets always as part of a transaction and not in a quick-and-dirty, direct way.

The concept can be transferred to memory operations performed in program as well. It can be used to replace critical sections protected by locks in a multi-threaded parallel program. Compared with critical sections, transactions have several advantages. First, programmers are liberated from guaranteeing the correctness and performance of their locking scheme. Next, shared data structures are guaranteed to be kept in consistency even in the event of a failure. Finally, transactions can be composed naturally, which make it much easier for developing composable parallel software.

4. IMPLEMENTATION DETAILS
4.1 Transactional memory designs
In this section, we will try looking into the internals of STM to have a better understanding about how an STM system is implemented to support this programming paradigm. Keep in mind
that this report covers STM only. However, there are other design choices, and it would be worth reviewing them briefly:

**Software TM** The systems are solely implemented by software, and therefore no additional hardware resources are needed. Only basic hardware instructions such as Compare-And-Swap currently available in most modern hardware designs are required. While offering the highest flexibility, it leads to overhead in excess of most users tolerance. There is a rich set of implementations for the moment: DSTM, RSTM, OSTM, McRT-STM, Bartik STM, WSTM, SXM, ASTM, TinyTM, Haskell STM, Ennals STM, AUTOLOCKER, TL, Shavit-STM, Ananian STM, Intel STM, Sun TL2, IBM STM, etc.

**Hardware TM** HTM systems can be purely implemented or supported by hardware. Although promising the best performance and strong atomicity, it suffers from several impediments: high implementation and verification costs; poor flexibility; and additional hardware overhead.

**Hybrid TM** Hybrid systems might be the most likely platform for the eventual adoption of TM by a wide audience, although the exact mix of hardware and software support remains unclear. A special case of the hybrid system is the hardware-accelerated STM. In this scenario, the transactional semantics are provided by STM, and hardware primitives are used only to speed up critical performance bottlenecks in the STM system.

**Others** By compromising some of the assumptions constrained by TM, it is possible to devise other design choices to overcome certain limitations incurred by aforementioned systems. For instance, interaction with non-transactional codes, including access to shared data from outside of a transaction (tolerating weak atomicity) and the use of locks inside a transaction (breaking isolation to make locking operations visible outside transactions) could be allowed.

Interested users can refer to an article that was published on the magazine *Queue* in 2008 to get more information regarding the designs [2].

### 4.2 Concurrency control mechanisms in DBMS

Since STM was originated from Databases research area, it would be meaningful to understand several mechanisms in DBMS to achieve concurrency control. They have been proved to be successful not only from the theoretical perspective but in large applications in industry. A number of commercial DBMS suggest that there are matured technologies to control concurrent transactions in a system. In principle, the implementations of STM follow the exact concepts used in DBMS.

We may classify the concurrency control (CC) mechanisms in three categories in a broad sense:

- **Optimistic CC** This assumes that although conflicts are possible, they will be very rare. Instead of locking every object every time that it is used, the system merely looks for indications that two transactions actually did try to modify the same object at the same time. If that evidence is found, then one transaction’s modifications are discarded and the transaction is informed. Usually, this may rollback the victim.

- **Pessimistic CC** This is called “pessimistic” because the system assumes the worst—it assumes that two or more transactions will want to modify the same object at the same time, and then prevents that possibility by locking the object, no matter how unlikely conflicts actually are.

- **Semi-optimistic CC** This combines the two approaches introduced earlier. It blocks operations in some situations, and does not block in other situations, while delaying checking conflicts to transaction’s end, as done with optimistic.

Many methods for concurrency control exist. Most of them can be implemented within either main category above. Major methods, which have each many variants, include:

- Two phase locking (2PL)
- Conflict (or serialization, serializability, precedence) graph checking
- Timestamp ordering (TO)
- Commitment ordering (or Commit ordering; CO)
- Multiversion concurrency control (MVCC)
- Index concurrency control (for synchronizing indexes, rather than data)

There are rich sources for details about the optimistic and pessimistic CC on the web. Some of them are mentioned in the references section. It would be a good start to visit pages on Wikipedia [3].

Optimistic CC is useful if the possibility for conflicts is very low—there are many objects but relatively few concurrent transactions, or very few modifications and mostly read-type operations. Additionally, it allows fast performance and high concurrency (access by multiple transactions), at the cost of occasionally refusing to write data that was initially accepted but was found at the last second to conflict with another transaction’s changes.

On the other hand, pessimistic CC commonly involves locks and requires overhead for every operation, whether or not two or more transactions are actually trying to access the same object. The overhead is small but adds up because every object that is to be modified requires a lock. Furthermore, every time that a transaction tries to access an object, the system must also check whether the requested object is already locked by another transaction.

Because of some desired features STM has to implement, it usually takes optimistic approach in a broad sense.

### 4.3 Internals of STM

Different implementations of software transactional memory systems make tradeoffs that impact performance. I recommend people who are deeply interested in internals of STM systems to refer to this research article published recently [4]. It will greatly help to understand a lot of fundamental concepts used in STM design and can reveal some details about many implementations that are currently available.

Here are the five categories that were used in the paper to classify various STM systems:

1. **Transactional granularity**

   Transactional granularity is the data store unit, through which a TM system detects conflicts. In general, STM systems have three granularity choices in reality implementation: word (or block), object and hybrid. With the word granularity, a shared word is possessed by no more than one transaction at any time. Word STM can get more concurrently access to data structures such as array, matrix etc. At the same time, it provides higher conflict detection accuracy. However, it leads to much more additional communication and associated metadata costs. These costs can cut down by increasing...
conflict detection granularity to block (multiword), although they bring false conflict when no real conflicts happen, injuring performance by making unnecessary transaction aborts.

2. Data organization

In STM system, the methods to organize transactional data in memory can be classified into two categories: In the first category, systems organize transactional data and ordinary data in separate memory structures. In these systems, a transaction must take actions to open a TM object first if it wants to access a concurrent object. If the access mode is READ, the same object body can be shared by multiple transactions at the same time. On the other hand, if the access mode is WRITE, a new version copy of the object is prepared for update and the new version copy of the object is only visible to the transaction until the transaction commits; In the second category, systems leave transactional data in ordinary memory structure and use a separate structure to store metadata which are used for transactions concurrency control. They refer transactional data by ordinary pointer directly thus reduce a map via object pointer. They are convenient for spatial access locality.

3. Version management

Version management is the mechanism to deal with the different versions of a logical data: the new updated versions from different transactions and the old version for rollback to the original data in case a transaction aborts. Generally, there are two kinds of version management: Lazy Version Management (LVM) and Eager Version Management (EVM): In STM systems with LVM, old version is remained in its original place and new versions are stored in a per-transaction buffer. Multiple transactions can concurrently access a shared object, with each of them keeping a private version of the object in store buffer. Hence, LVM allows concurrent transactional read and write for the same logical data; Compared with LVM, EVM reduces the copy cost in LVM, because the new version data is stored in the old versions address, thus only a new version can be stored. However, it prevents other transactions to read a modified uncommitted object, thus limits the possible concurrency.

4. Conflict detection

Generally, there are three type of conflict detection: Eager Conflict Detection (ECD), Lazy Conflict Detection (LCD) and Hybrid Conflict Detection (HCD). The ECD detects conflicts when a transaction wants to access memory while LCD detects conflicts when a transaction is about to commit updates. ECD always works with EVM, since it is necessary to make sure that only one transaction can write a new version to a logical data. Thus, the system must detect conflicts first. Similarly, LCD usually works with LVM, since all updates are private and invisible to others, which do not need conflict detection before committing. Some HTM systems provide the combination of LVM and ECD, which is rarely used in STM systems. Some STMs use HCD, which combines ECD and LCD.

5. Synchronization

Synchronization is the mechanism to guarantee that a transaction attempting to access a logical data will finally finish its work. In general, there are two types of concurrent control: Blocking Synchronization (BS) and Non-blocking Synchronization (NS). The BS is a conventional form of synchronization, constructed with locks, monitors or semaphores. In order to keep consistency, BS forces multiple threads to access critical sections exclusively. The NS has been classified into three main categories based on their assurances for forward progress: Wait-freedom, Lock-freedom and Obstruction freedom. Early researchers for STM systems focus on non-blocking data structures with NS to guarantee forward progress.

5. MOTIVATION FOR STM

No mainstream language, compiler, or library currently supports STM. So why should we as programmers care about it? I think the best answer is that it could make our jobs easier in the future. If you keep an eye on the technology now, you’ll have an easier time transitioning to this new “STM paradigm” when it matures, which it hopefully will.

Two areas where I think STM will be especially helpful are balancing concurrency and simplicity and designing composable programs. Let’s look at each of these.

5.1 Balancing concurrency and simplicity

This has been a theme running throughout the class: there are usually tradeoffs between code that is highly concurrent and code that is easy to reason about.

As a general example, at the beginning of the semester we considered a simple design for a concurrent server, where every incoming request caused a new worker thread to fire off to handle it. That was simple to reason about because you didn’t have to think much about the state of the workers; the worker is created when a request comes in and is destroyed when the corresponding response goes out. This turned out not to be a great design, since the “one thread per request” model doesn’t scale well on mainstream architectures. So we switched to a thread pool design, where we fire off a fixed number of worker threads when the server boots up, and have them contend for incoming requests. But this meant that we had to put a lot of logic into the scheduler, since the workers that it oversees could be in several different states, e.g. reading a request, writing a response, fetching something from the cache, etc.

The same sort of problem is apparent with concurrent data structures. The easiest way to turn a sequential data structure into a concurrent one is to create one big lock and acquire and release it for every operation on the data structure. That doesn’t seem fair to the computer, though, since most data structures are made up of lots of small parts. If two threads want to perform operations on two different parts of the data structure concurrently, that ought to be allowed. One response is to decrease the granularity of the locks. The tradeoff here is that now we have to think about deadlocks, which could happen if threads don’t acquire locks in the same order. (We also have to make sure the time a thread spends acquiring and releasing these tiny locks doesn’t cancel out the benefits of the higher concurrency.)

If we want to go even further, we realize that even an operation on a single part of the data structure is usually comprised of several mini-operations, such as pointer updates. If two threads want to operate on the same part of the data structure concurrently, that ought to be allowed, so long as the mini-operations don’t interfere with each other. This kind of thinking leads to the lock-free data structures, which are technically impressive but very hard to reason about. The lock-free queue we saw, for example, allowed operations to return successfully while leaving the queue in a “half-finished” state. Subsequent operations would move the queue to a
fully finished state, but it's not easy to understand from the code why this always happens.

STM can be thought of as a balancing act between concurrency and simplicity. Instead of designing completely new programs to take advantage of concurrency, we give the computer something that resembles sequential or very basic concurrent code, and we ask the computer to manage concurrency in this code to the best of its abilities. (This general situation turns up in plenty of other areas. For example, instead of performing loop optimizations by hand at the source level, something we could certainly do, we ask the compiler to perform them.) Even though current STMs may not be able to beat custom-designed concurrent code, the hope is that they eventually will be decent enough so that custom-designing concurrent code is not a good use of a programmer's time. I'm optimistic about this because the process has happened several times before. That's basically what compilers are, for instance.

5.2 Composability

Besides this general problem of tradeoffs between concurrency and simplicity, there is a more specific issue with locks themselves. Locking is just one tool for implementing concurrency; there are others that we haven't looked at, for example message passing. There's no reason to expect locks to be an elegant answer for every concurrent situation, and indeed there is a very important situation where locks don't feel like the right tool: composition.

If you've ever used a language that directly supports function composition, currying, or even simple lambda expressions (Python and the upcoming C++ standard have this last feature), you know how important these techniques are. Even in C-like languages where functions aren't quite as flexible as other data, the idea of building large, powerful features out of small, simple ones is fundamental.

An issue with this sort of "composition" in C is that you often have to force it along manually; you can't take working components and string them into a whole that works with no further effort on your part. Here's an example: some system calls are typically used in a sequence to achieve a desired state, e.g. socket/bind/listen/accept to set up a listening socket on a server. But you can't just write

\[
\begin{align*}
&\text{socket(...)}; \\
&\text{bind(...)}; \\
&\text{listen(...)}; \\
&\text{accept(...)}; \\
\end{align*}
\]

Any one of these functions could fail, so you have to write something more like

\[
\begin{align*}
\text{if (socket(...)==-1) } & \text{ /* inspect and handle errno */} \\
\text{else if (bind(...)==-1) } & \text{ /* inspect and handle errno */} \\
\text{else if (listen(...)==-1) } & \text{ /* inspect and handle errno */} \\
\text{else if (accept(...)==-1) } & \text{ /* inspect and handle errno */} \\
\end{align*}
\]

which easily doubles the amount of work you have to do, partially negating the convenience of composition in the first place.

Exceptions in C++ are a step in the right direction, since instead of doing the error handling after each function call, you can throw an exception and concentrate all the error handling in one place. (You can sort of do this in C with the setjmp and longjmp standard library functions, but it is cumbersome.) This is getting close to what we would like as programmers. We would like to tell the computer, "execute this block of code and if anything goes wrong, either start over and try it again or tell us". But note that even with C++, we still have to throw the exceptions ourselves.

It should be clear that the programmer overhead involved with supervising composition in general applies to composition of synchronization operations in particular. Any synchronization that might fail, for example a pthread_mutex_trylock or a pthread_cond_timedwait, has to be checked and dealt with. Data structures and algorithms that synchronize internally can fail in interesting ways. What do you do with an object that failed to be enqueued in a concurrent bounded queue? Spin until the enqueue succeeds? Put it back where it came from (an operation that might also fail)?

Even in the absence of error conditions, composition is a hazard for locks. If you want to perform a sequence of operations on a data structure that locks internally, you might be acquiring and releasing the lock multiple times, where only the outermost acquire and release would suffice. (This is what C standard I/O, fread and fwrite for example, does.) Perhaps you should change the interface of the data structure to expose the locks to programmers who know what they are doing, and also expose unlocked variants of the operations? The C standard I/O library has a bunch of variants that do exactly this. Instead of writing

\[
\begin{align*}
&\text{fopen(...);} \\
&\text{fread(...);} \\
&\text{fwrite(...);} \\
&\text{fclose(...);} \\
&\text{you can write} \\
&\text{fopen(...)}; \\
&\text{flockfile(...)}; \\
&\text{fread_unlocked(...)}; \\
&\text{fwrite_unlocked(...)}; \\
&\text{funlockfile(...)}; \\
&\text{fclose(...)}; \\
\end{align*}
\]

This approach gives the client programmer quite an opportunity to mess up. (Think of the trouble a programmer with only a vague understanding of deadlocks could get into.) It also freezes your choice of synchronization mechanism, whereas if you do all the synchronization internally, you are free to change it whenever.

Again, what we would like as programmers is a way to demarcate critical sections of code and tell the computer (the compiler, the runtime environment, a library—something): "allow concurrent access to this section, but if a conflict arises, roll the state back to the beginning, and either try again or inform us of the failure". And we would like all this to be as automatic as possible.

Ideally, we would just demarcate the critical section similar to a pthread_mutex_lock/pthread_mutex_unlock pair, and the computer would figure out the rest, including how to detect a conflict and how to roll back the state properly.

This is pretty much what a transaction in STM is: a block of code that looks like a coarse-locked critical section, but that tries to handle concurrency as automatically as possible.

6. OVERVIEW OF CURRENT STM IMPLEMENTATIONS

STM is an idea, not a specific technology, and so it’s not surprising that implementations target different points along the build path. At one extreme, we could put STM into the language itself. The only language I know of that does this is Clojure [5]. (This language doesn’t appear to have a specification, but it also doesn’t have more than one implementation, so it’s fair to say it’s in the lan-
7. PROGRAMMING WITH TINYSTM

TinySTM [10] is one of the more mature STM libraries from the client programmer’s point of view. It’s a plain C library, and its only dependency is something called libatomic-ops, which is available in the Debian/Ubuntu repositories or on the web [11]. It’s well documented, and the makefile gives you a lot of control over the strategies for detecting conflicts in a transaction and so forth, so it’s fun to play around with.

Let’s see how we might convert a simple concurrent data structure to use TinySTM. Figure 1 shows the enqueue method for a concurrent queue implemented as a linked list with a single coarse lock.1 We lock the lock (line 3) and allocate a new node (line 4). If the back pointer doesn’t point to anything, i.e., the queue is empty (line 5), the queue will now have one element, and both the front and back pointers will point to it (line 6). Otherwise we swing the new node onto the end of the back pointer (line 8) and point the back pointer to the new node (line 9). Finally we unlock the lock (line 11) and return.

Figure 2 shows the enqueue method for a concurrent queue implemented as a linked list using TinySTM transactions. I’ve defined a few macros of my own to make TinySTM a little easier to use; I’ll walk through these definitions in a moment. The main idea is that we start a transaction (line 3), allocate a new node (line 4), and check if the queue is empty (lines 6–7). If it is, we point both front and back pointers to the new node (lines 8–9). Otherwise we swing the new node onto the end of the back pointer (line 10) and point the back pointer to the new node (line 11). Finally we commit the transaction (line 12) and return. Thus, the logic of this method is identical to that of the coarse-locked queue.

The main difference is syntactic, because, unfortunately, you have to explicitly tell TinySTM which loads and stores inside a transaction to monitor. The wrappers.h header provides the functions stm_load_ptr and stm_store_ptr, and my LOAD_PTR and STORE_PTR macros are wrappers around them, just to avoid having to do the type casting each time:

```c
#define LOAD_PTR(addroptr) \n    stm_load_ptr((volatile void **)addroptr)
#define STORE_PTR(addroptr, value) \n    stm_store_ptr((volatile void **)addroptr, (void *) value)
```

You also have to be explicit about allocations inside a transaction, since TinySTM has to know how to undo them in the event that a transaction aborts. Thus in line 4, we use stm_malloc (defined in the mod_mem.h header) instead of malloc or operator new.

Let’s see what the START and COMMIT macros are doing (figure 3). The way that TinySTM implements flow of control around a transaction is on top of the old setjmp and longjmp standard library functions. The START macro just sets up a jump buffer pointing at the beginning of the transaction (lines 4–6), so that if the transaction aborts, it will automatically retry. (If we don’t want this behavior, TinySTM allows us to pass a null jump buffer to stm_start. In this case we have to manually check the return from stm_commit to see whether the transaction succeeded.) The id parameter of START is just an identifier for the present transaction that might be helpful for debugging, and the ro parameter is a hint about whether the transaction is read-only or not.

There are a couple of other things you have to do to set up TinySTM. You have to initialize the library at the outset with stm_init, and you have to initialize each thread that will perform transactions

---

1This implementation ignores a few issues for the sake of clarity. For example, we probably want to allocate the new node before entering the critical section. Also, if the queue is empty, we may want to signal on a condition variable associated with the lock in order to wake up a thread that might be doing a blocking dequeue.
```c
#define RO 1
#define RW 0
#define START(id, ro)
{ 
  sigjmp_buf *_e = stm_get_env();
  stm_tx_attr_t _a = {id, ro};
  sigsetjmp(*_e, 0);
  stm_start(_e, &_a);
#define COMMIT stm_commit(); }
```

Figure 3: Convenience macros for TinySTM transactions.

(including the “main” thread that called stm_init, if need be) with stm_init_thread. There are corresponding thread and global shutdown functions, so the whole library could rather easily be wrapped in a C++ class with constructors and destructors, though I stuck to macros here for the sake of clarity.

I should note that the designers of TinySTM also have a tool, the Dresden TM Compiler [10], that can eliminate the tedium of explicitly coding transactional loads and stores. It is basically a source-to-source compiler that transforms code within transaction boundaries into explicit TinySTM calls where appropriate. But because it requires modifications to the host compiler, I didn’t use it.

Finally, a word of caution: TinySTM presents a pretty good interface to the programmer, but the internal implementation may not be all that mature. Internally, it uses a hierarchy of locks to protect the portions of the address space that may be shared; although the locks can expand and contract at runtime, depending on the extent of the concurrent code, the initial number of locks is huge, $2^{20}$ [12]. My informal tests suggest that the TinySTM concurrent queue discussed above runs something like an order of magnitude slower than the coarse-locked concurrent queue, and its performance against more sophisticated structures, like a lock-free concurrent queue, is probably worse. I would definitely recommend using the mod_stats TinySTM module, which gives you access to performance statistics. Again in my informal tests, I found that the TinySTM concurrent queue was incurring something like 80 aborted transactions for every successful transaction.

8. REFERENCES