ByteSTM: Virtual Machine-level Java Software Transactional Memory

Mohamed Mohamedin
ECE Dept., Virginia Tech, Blacksburg, VA, USA
mohamedin@vt.edu

Binoy Ravindran
ECE Dept., Virginia Tech, Blacksburg, VA, USA
binoy@vt.edu

Abstract

We present ByteSTM, a virtual machine-level Java STM implementation. ByteSTM implements two STM algorithms, TL2 and RingSTM, and transparently supports implicit transactions. Program bytecode is automatically modified to support transactions. Being implemented at the VM-level, it accesses memory directly and uses absolute memory addresses to uniformly handle memory, and avoids Java garbage collection by manually managing memory for transactional metadata. ByteSTM uses field-based granularity. Our experimental studies reveal throughput improvement over other Java STMs in the range of 13%–70% on micro-benchmarks and 10%–60% on macro-benchmarks.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming–parallel programming; D.3.3 [*Programming Languages*]: Language Constructs and Features–concurrent programming structures; D.3.4 [*Programming Languages*]: Processors–run-time environments

General Terms Algorithms, Experimentation, Languages.

Keywords software transactional memory (STM), transactions, concurrency, atomicity, run-time, virtual machines

1. Introduction

Lock-based synchronization is the most widely used synchronization abstraction. Coarse-grained locking is simple to use, but limits concurrency. Fine-grained locking permits greater concurrency, but has low programmability: programmers must acquire only necessary and sufficient locks to obtain maximum concurrency without compromising safety, and must avoid deadlocks when acquiring multiple locks. Moreover, locks can lead to livelocks, lock-convoying, and priority inversion. Perhaps, the most significant limitation of lock-based code is its non-composability [20].

Transactional Memory (TM) is an alternative synchronization abstraction that promises to alleviate these difficulties. With TM, code that read/write shared memory objects is organized as *memory transactions*, which speculatively execute, while logging changes made to objects–e.g., using an undo-log or a write-buffer. Two transactions conflict if they access the same object and one access is a write. A contention manager resolves the conflict by aborting one and allowing the other to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started, after rolling-back the changes–e.g., undoing object changes using the undo-log (eager), or discarding the write buffer (lazy). In addition to a simple programming model (locks are excluded from the programming interface), TM provides performance comparable to lock-based synchronization [37] and is composable.

TM has been proposed in hardware (HTM [19]), in software (STM [38]), and in combination (HybridTM [26]). HTM has the lowest overhead, but transactions are limited in space and time. STM does not have such limitations, but has higher overhead. HybridTM avoids these limitations.

Thread A

```
1 atomic{
2   for (int i=0; i<1000;
       i++)
3     counter++;
4 }
```

Thread B

```
1 atomic{
2   for (int i=0; i<1000;
       i++)
3     counter++;
4 }
```

Figure 1. Example of implicit transaction language support. If `counter` is initialized to zero, the final value will be 2000.

Figure 1 shows an example TM code. The example uses the `atomic` keyword, which implicitly creates a transaction for the enclosed code block.

1.1 STM Implementations

Given the hardware-independence of STM, which is a compelling advantage, we focus on STM. STM implementations can be classified into three categories: *library-based*, *compiler-based*, and *virtual machine-based*. Library-based STMs add transactional support without changing the underlying language, and can be further classified into: those that use *explicit* transactions [16, 21, 33, 44] and those that use *implicit* transactions [10, 25, 36]. Explicit transactions are difficult to use. They support only transactional objects, and hence cannot work with external libraries. Implicit transactions, on the other hand, use modern language features (e.g., Java annotations [40]) to mark sections of the code

Feature	Deuce [25]	JVSTM [10]	ObjectFabric [33]	AtomJava [22]	DSTM2 [21]	Multiverse [44]	LSA-STM [36]	Harris and Fraser [18]	Atomos [12]	Transactional monitors [45]	ByteSTM
Implicit transactions	✔	✔	✖	✔	✖	✖	✔	✔	✔	✔	✔
No instrumentation	✖	✖	✔	✔	✔	✔	✖	✔	✔	✖	✔
All data types	✔	✖	✖	✔	✖	✖	✖	✔	✔	✔	✔
External libraries	✔	✖	✖	✔ [1]	✖	✖	✖	✔	✖ [2]	✔	✔
Unrestricted atomic blocks	✖	✖	✔	✔	✔	✔	✖	✔	✔	✔	✔
Direct memory access	✔ [3]	✖	✖	✖	✖	✖	✖	✔	✔	✖	✔
Field-based granularity	✔	✖	✖	✖	✖	✖	✖	✖	✖	✖	✔
No GC overhead	✔ [4]	✖	✖	✖	✖	✖	✖	✔	✔	✖	✔
Compiler support	✖	✖	✖	✔	✖	✖	✖	✔	✔	✔	✔ & ✖ [5]
Strong atomicity	✖	✖	✔	✔	✖	✖	✖	✖	✔	✖	✖
Closed/Open nesting	✖	✔	✔	✖	✖	✖	✖	✖	✔	✖	✖
Conditional variables	✖	✖	✖	✖	✖	✖	✖	✖	✔	✖	✖

[1] Only if source code is available.
[2] It is a new language, thus no Java code is supported.
[3] Using non-standard library.
[4] Uses object pooling, which partially solves the problem.
[5] ByteSTM can work with or without compiler support.

Table 1. Comparison of Java STM implementations.

as atomic. Instrumentation is used to add transactional code transparently to the atomic sections. Some implicit transactions work only with transactional objects [10, 36], while others work on any object and support external libraries [25].

Compiler-based STMs (e.g., Intel C++ STM compiler [14], AtomJava [22]) support implicit transactions transparently by adding new language constructs (e.g., `atomic`). The compiler then generates transactional code that calls the underlying STM library. Compiler-based STMs can optimize the generated code and do overall program optimization. On the other hand, compilers need the source code to support external libraries. With managed run-time languages, compilers alone do not have full control over the VM. Thus, the generated code will not be optimized and may contradict with some of the VM features like the garbage collector (GC).

VM-based STMs, which have been less studied, include [2, 12, 18, 45]. In [18], STM is implemented in C inside the JVM to get benefits of the VM-managed environment. This STM uses an algorithm that does not support the opacity correctness property [17]. This means that inconsistent reads may occur before a transaction is aborted, causing unrecoverable errors in an unmanaged environment. [12] presents a new programming language based on Java, called Atomos, and a VM to run it. Standard Java synchronization (i.e., `synchronized`, `wait`/`notify`) is replaced with transactions. However, transactional support is based on HTM.

Library-based STMs are largely based on the premise that it is better not to modify the VM or the compiler, to promote flexibility, backward compatibility with legacy code, and easiness to deploy and use. However, this premise is increasingly violated as many require some VM support or are being directly integrated into the language and thus the VM. Most STM libraries are based on annotations and instrumentation, which are new features in the Java language. For example, Deuce STM library [25] is based on a non-standard proprietary API (i.e., `sun.misc.Unsafe`), which makes it incompatible with other JVMs. Moreover, programming languages routinely add new features in their evolution for a whole host of reasons. Thus, as STM gains traction, it is natural that it will be integrated into the language and the VM.

Implementing STM at the VM-level allows many opportunities for optimization and adding new features. For example, the VM has direct access to memory. Thus, writing values back to memory is easier, faster, and can be done without using reflection (which usually degrades performance). The VM also has full control of the GC, which means that, the GC's potentially degrading effect on STM performance can be controlled or even eliminated by manually allocating and recycling the memory needed for transactional support. Moreover, if TM is supported using HTM (as in [12]), then VM is the only appropriate level of abstraction (from a performance standpoint) to exploit that support. Otherwise, if TM is supported at a higher level, the GC will abort transactions when it interrupts them, and language synchronization semantics will contradict with transactional semantics. Also, VM memory systems typically use a centralized data

structure, which increases the number of conflicts, degrading performance [9].

1.2 Contributions

Motivated by these observations, we design and implement a VM-level STM: ByteSTM (Section 2). ByteSTM implements two algorithms: TL2 [15] and RingSTM [39]. It writes directly to memory without using reflection (unlike [21]) or a non-standard library (unlike [25]). ByteSTM uniformly handles all variable types, using the address and number of bytes as an abstraction. It eliminates the GC overhead, by manually allocating and recycling memory for transactional metadata. ByteSTM uses field-based granularity, which scales better than object-based or word-based granularity, and has no false conflicts due to field granularity.[1]

ByteSTM supports implicit transactions in two modes: compiler mode and direct mode. The compiler mode requires compiler support and works with the `atomic` new keyword. The direct mode works with standard Java compilers by calling (special) static methods to start and end a transaction. A transaction can surround any block of code, and is not restricted to methods only. Memory bytecode instructions (i.e., memory load/store operations) that are reachable from a transaction are translated so that the resulting native code executes transactionally. Thus, no instrumentation is required. ByteSTM works with all data types, not just transactional objects, and thereby supports external libraries. Our current implementation does not support nesting, strong atomicity, or conditional variables; these are planned as future work.

Table 1 distinguishes ByteSTM from other STM implementations. Each row describes an STM feature, and each column describes an STM implementation. The table entries describe the features supported by the different STMs. (Section 4 summarizes competitor STMs.)

We conducted experimental studies, comparing ByteSTM with other Java STMs including Deuce [25], Object Fabric [33], Multiverse [44], DSTM2 [21], and JVSTM [10] (Section 3). Our results reveal that, ByteSTM improves transactional throughput over others by 13% to 70% on micro-benchmarks, and by 10% to 60% on macro-benchmarks. The overall average improvement is 30%.

ByteSTM is open-sourced and is publicly available at `hydravm.org/bytestm`. We hope this encourages replication of our results and further research in this problem space.

2. Design and Implementation

ByteSTM is built by modifying the JikesRVM [4] version 3.1.0. In ByteSTM, bytecode instructions can run in two modes: transactional and non-transactional. The visible modifications to the VM users are very limited: two new instructions are added (`xBegin` and `xCommit`) to the VM bytecode instructions. These two instructions will need compiler modifications to generate the correct bytecode when the *atomic* blocks are translated. Also, the compiler should handle the new keyword `atomic` correctly. However, in order to eliminate the need for a modified compiler, a simpler workaround is used, which is calling the static method `stm.STM.xBegin()` to begin a transaction, and `stm.STM.xCommit()` to commit the transaction. These two methods are defined empty and static in the class `STM` in `stm` package.

ByteSTM is implicitly transactional: the program only specifies the start and the end of the transaction and all memory operations (loads and stores) inside these boundaries are implicitly transactional. This simplifies the code inside the atomic block and also eliminates the need for making a transactional version for each memory load/store instruction, thereby keeping the number of added instructions minimal. When `xBegin` is executed, the thread enters in transactional mode. In this mode, all writes are isolated and the execution of the instructions is speculative until `xCommit` is executed. At that point, the transaction is compared against other concurrent transactions for a conflict. If there is no conflict, the transaction is allowed to commit and at this point (only), all the transaction modifications become visible to the outside world. If the commit fails, all the transaction modifications are discarded and the transaction restarts from the beginning.

In the current implementation of ByteSTM, we use the Jikes Baseline Compiler [42], which does not support many optimizations (e.g., inlining, local optimization, control optimization, global optimization). This compiler is simply used to translate from bytecode instructions to native code, one by one. For each memory load/store instruction, native code is generated, which checks whether the thread is running in transactional mode or non-transactional mode. Thus, the instruction execution continues transactionally or non-transactionally. (Our immediate future work includes replacing the Jikes Baseline Compiler with the Jikes Optimizing Compiler [43].)

Modern STMs [10, 25, 36] use automatic instrumentation. Java annotations are used to mark methods as atomic. The instrumentation engine then handles all code inside atomic methods and modifies them to run as a transaction. This conversion does not need the source code and can be done offline or online. Instrumentation allows, for the first time, using external libraries – i.e., code inside a transaction can call methods from an external library, which may modify program data [25].

No instrumentation is required in ByteSTM. This means lesser overhead than online instrumentation. Any code that is reachable from within a transaction is compiled to native code with transactional support. Classes/packages that will be accessed transactionally are input to the VM by specifying them on the command line. Then, each memory operation in these classes is translated so that it first

[1] False conflicts may still occur due to other implementation choices, e.g., TL2 lock table [15], read/write signatures [39].

checks if the thread is running in transactional mode. If so, it runs transactionally. Otherwise, it runs regularly (i.e., non-transactionally). Although doing such a check with every memory load/store operation increases overhead, our results show significant throughput improvement over competitor STMs (see Section 3).

Atomic blocks can be used anywhere in the code (either using the `atomic` keyword or by calling `xBegin` and `xCommit`). It is not necessary to make a whole method atomic; any block can be atomic. External libraries can be used inside transactions without any change.

Memory access is monitored at the field level, and not at the object level. Field-based granularity scales well and eliminates false conflicts resulting from two transactions changing different fields of the same object.

2.1 Metadata

Working at the VM level allows changing the thread header without modifying the program code. For each thread that executes transactions, metadata added include the read signature, the write signature, the write buffer, and the start time. These metadata is added to the thread header and is used by all transactions executed in the thread. Figure 2 shows a thread header example with added metadata. Since each thread executes one transaction at a time, there is no need to create new data for each transaction, allowing reuse of the metadata. Also, accessing a thread's header is faster than Java's `ThreadLocal` abstraction.

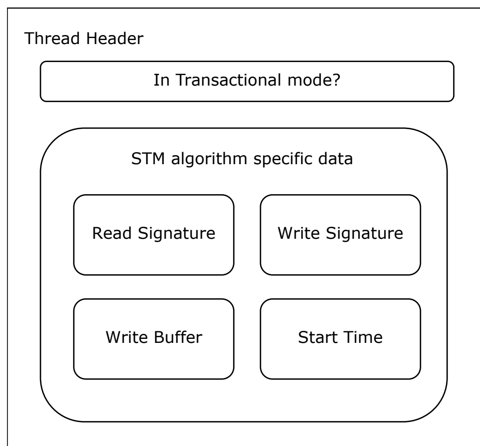


Figure 2. A thread header example with added metadata.

2.2 Memory Model

At the VM-level, the memory address of each field of an object can be easily obtained. As mentioned before, ByteSTM is not object-based, it is field-based. The address of each field is used to track memory reads and writes. A conflict occurs only if two transactions modified the same field of an object. Static objects are also supported.

In Java, arrays are objects. ByteSTM tracks memory accesses to arrays at the element level. That way, unnecessary aborts are eliminated. Moreover, no reflection is needed and data is written directly to the memory, as a memory address is already available at each load and store.

Absolute memory address is used to unify different addressing mechanisms used in Java. An object instance's field's absolute address equals the object's base address plus the field's offset. A static object's field's absolute address equals the global static memory space's address plus the field's offset. Finally, an array's element's absolute address equals the array's address plus the element's index in the array (multiplied by the element's size).

To optimize memory access, memory access is handled in the raw format. This means that the address of a memory location and the number of bytes at that address are all the information needed to access that location, irrespective of the data type of the location. This abstract view simplifies how the read-set and the write-set are handled. At a memory load, all information needed to track the read is the memory address of the read location. At memory store, the memory address, the new value, and the size of the value are the information used to track the write. When data is written back to memory, the write-set information (address, value, and length of the location) is used to store the committed values correctly. This abstraction also simplifies the code, as there is now no need to differentiate between different data types, as they are all handled as a sequence of bytes in the memory. The result is simplified code that handles all the data types, and smaller number of branches (no type checking), yielding faster execution.

2.3 Write-set Representation

We found that using a complex data structure to represent read-sets and write-sets affects performance. Given the simplified raw memory abstraction used in ByteSTM, we decided to use simple arrays of primitive data types. This decision is based on two reasons. First, array access is very fast and has access locality, resulting in better cache usage. Second, with primitive data types, there is no need to allocate a new object for each element in the read/write set. (Recall that an array of objects is allocated as an array of references in Java, and each object needs to be allocated separately. Hence, there is a large overhead for allocating memory for each array element.) Even if object pooling is used, the memory will not be contiguous since each object is allocated independently in the heap.

Using arrays to represent the write-set means that the cost of searching an n-element write-set is $O(n)$. For $n \leq 10$ (which is the case in micro-benchmarks [24]), this is acceptable, and was found to be faster than using hashing, given the overhead of the standard Java hash table (which uses linked-lists for bucket overflows and supports only objects).

To obtain the benefits of arrays and the speed of hashing, open-addressing hashing with linear probing is used. We used an array of size 2^n, which simplifies the modulus calculation.

The hash function that we used is simple, which simply removes the upper bits from the memory address using bitwise *and* operation (which is equivalent to calculating the modulus of the address): *address AND mask = address MOD arraySize*, where *mask = arraySize* - 1. For example, if *arraySize = 256*, then *hash(address) = address AND 0xFF*. This hashing function is very efficient with addresses, as the collision ratio is very small. When a collision happens, there is always an empty cell after the required index because of the memory alignment gap (so linear probing will give good results). This way, we have a very fast and efficient hashing function that adds very little overhead to each array access, enabling $O(1)$-time searching and adding operations on large write-sets.

Iterating over the write-set elements by cycling through the sparse array elements is not efficient. We solve this by keeping a contiguous log of all the used indices, and then iterating on the small contiguous log entries.

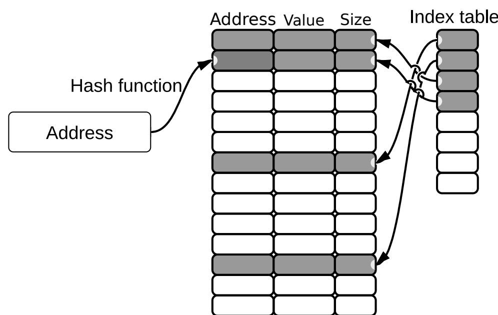


Figure 3. ByteSTM's write-set using open address hashing.

Open addressing has two drawbacks: memory overhead and rehashing. These drawbacks can be controlled by choosing the array size such that the number of rehashing is reduced, while minimizing memory usage. Figure 3 shows how ByteSTM's write-set is represented using open-addressing.

2.4 Atomic Blocks

ByteSTM supports atomic blocks anywhere in the code. When `xBegin` is executed, the current program state is saved. If a transaction is aborted, the saved state is restored and the transaction can restart as if nothing has changed in local variables – i.e., similar to `setjmp`/`longjmp` in C. This technique simplifies the handling of local variables since there is no need to monitor them. Note that, at the VM level, full control of the program state is possible.

2.5 Garbage Collector

One major drawback of building an STM for Java (or any managed language) is the GC [29]. STM uses metadata to keep track of transactional reads and writes. This requires allocating memory for the metadata and then releasing it when not needed. Frequent memory allocation (and implicit deallocation) forces the GC to run more frequently to release unused memory, increasing the overhead on STM operations.

Some STMs have tried to solve this problem by reducing memory allocation and recycling the allocated memory [25]. For example, object pooling is used to reduce the pressure on the memory system and improve performance in [25], wherein objects are allocated from, and recycled back to a pool of objects (with the heap used when the pool is exhausted). However, allocation is still done through the Java memory system, and the GC will continue to check if the pooled objects are still referenced.

Since ByteSTM is integrated into the VM, its memory allocation and recycling is done outside the control of the Java memory system: memory is directly allocated and recycled. STM's memory requirement, in general, has a specific lifetime. When a transaction starts, it requires a specific amount of metadata, which remain active for the transaction's duration. When the transaction commits, the metadata is recycled. Thus, manual memory management does not increase the complexity or overhead of the implementation.

The GC causes another problem for ByteSTM, however. ByteSTM stores intermediate changes in a write buffer. Thus, the program's newly allocated objects will not be stored in the program's variable. The GC scans only the program's stack to find objects that are no longer referenced. Hence, it will not find any reference to the newly allocated objects and will recycle their memory. When ByteSTM commits a transaction, it will thus be writing a *dangling* pointer. We solve this problem by giving the GC a list of all intermediate objects in a transaction's write buffer, so that the GC will not "touch" them.

2.6 STM Algorithms

As mentioned before, ByteSTM implements two STM algorithms: TL2 [15] and RingSTM [39]. Our rationale for selecting these two algorithms is that, they are the best performing algorithms reported in the literature. Additionally, they cover different points in the tradeoff space: TL2 is effective for long transactions, moderate number of reads, and scales well with large number of writes, while RingSTM is effective for transactions with high number of reads and small number of writes. We briefly overview these algorithms for completeness.

With TL2, a global lock table consisting of versioned locks is used to synchronize access to shared locations. A global clock is also used to tag each transaction with its starting time (version). A transaction's writes are buffered in a redo log. Each transaction has a read-set. A transaction validates its read-set by checking the corresponding lock in the lock table. If the lock is acquired, or if it has a larger version than the transaction's version, then the transaction aborts. At commit, a validation is done again for all read-set entries. If the transaction is valid, then the current transaction's redo log entries are locked. If all locks are acquired successfully, then the transaction writes-back the redo log values to mem-

ory, increments the global clock, and finally updates the acquired locks' versions with the new clock value and releases the locks.

With RingSTM, a transaction's writes are buffered in a redo log. Each transaction has a read signature and a write signature (i.e., Bloom filter [8]) that summarize all read locations and written locations, respectively. A transaction validates its read-set by intersecting its read signature with other concurrent committed transactions' write signatures in a ring. The ring is a circular buffer that has all committed transactions' write signatures. At commit, a validation is done again. If the transaction is valid, then the current transaction's write signature is added to the ring using a single Compare-And-Swap operation. If it is successfully added to the ring, then the transaction is committed, and it writes-back the redo log values to memory.

2.7 Limitations

Currently, ByteSTM does not support running irrevocable operations (e.g., I/O operations) inside a transaction. One way to support such operations is to automatically convert a transaction to an irrevocable one when it performs any irrevocable action. Irrevocable transactions are guaranteed to commit successfully by executing them non-concurrently, of course, at the expense of reduced throughput. (Note that none of the Java STMs in Table 1 support irrevocable transactions.)

ByteSTM does not support nesting. But the mechanism of storing transaction's metadata in the thread header can be easily extended to support linear nesting (i.e., all child transactions run in the same thread of the parent) [30]. For example, the thread header can hold a tree representing parent/child relationship, and each node may hold transaction metadata. Each transaction can then access its metadata and its parent's metadata directly from the thread header. For parallel nesting (i.e., each child transaction runs in its own thread) [3, 6], a global data structure, where a child can find its parent's metadata, can be added.

Currently, ByteSTM works only with non-moving GC (e.g., mark-and-sweep [27]), where the object address does not change. One way to support moving GC [23] is to use two fields to represent a field address: the field's object's address and the field's offset. The GC would have access to all transactions' read/write sets. At the end of its garbage collection cycle, the GC scans all object addresses and updates them the same way it updates regular object references (e.g., using forwarding pointers). The same technique can be used for array elements, but the field's offset will hold the element index multiplied by the element size.

Adding support for irrevocable operations, nesting, and moving GC is future work.

3. Experimental Results

3.1 Test Environment

We conducted our experiments on a 48-core machine, which has four AMD Opteron™ Processors (6164 HE), each with 12 cores running at 1700 MHz, and 16 GB of memory. The machine runs Ubuntu Linux Server 10.04 LTS 64-bit. JikesRVM version 3.1.0 is used to run all experiments. We configured it to run using the Jikes Baseline compiler and mark-and-sweep GC, which match ByteSTM configurations.

The competitor STMs include Deuce [25], ObjectFabric [33], Multiverse [44], DSTM2 [21], and JVSTM [10]. (See Section 4 for a discussion on these STMs.) Note that Deuce uses a non-standard proprietary API (i.e., `sun.misc.Unsafe`), which is not fully supported by JikesRVM. To run Deuce atop JikesRVM, we therefore added necessary methods to JikesRVM `sun.misc.Unsafe` implementation including `getInt`, `putInt`, `getByte`, `putByte`, `getDouble`, `putDouble`, etc.

Since some of these competitor STMs use different algorithms (e.g., Multiverse uses a modified version of TL2; JVSTM uses a multi-version STM algorithm) or different implementations, a direct comparison between them and ByteSTM has some degree of unfairness. This is because, such a comparison includes many combined factors – e.g., the TL2 implementation in ByteSTM is similar to Deuce's TL2 implementation, but the write-set and memory management are different. This makes it difficult, in general, to conclude that ByteSTM's (potential) performance gain is exclusively due to implementing STM at the VM-level. Thus, we implemented a non-VM version using TL2 and RingSTM algorithms as Deuce plug-ins. Comparing between ByteSTM as an STM at the VM level with such a non-VM implementation reduces the number of factors in the comparison.

The non-VM implementation was made as close as possible to the VM one. The same open-addressing hashing write set is used. The absolute address is replaced by the field's object's reference and the field's offset. A large read-set and write-set are used so that they are sufficient for the experiment without requiring extra space. These sets are recycled for the next transactions. This way, only a single memory allocation is needed and the GC overhead is minimal. We used Deuce for this non-VM implementation, since it has many of ByteSTM's features. For example, it can directly access memory and uses field-based granularity. Moreover, it achieved the best performance among all STM competitors (see results later in this section). We used offline instrumentation to eliminate the online instrumentation overhead.

Our test applications include both micro-benchmarks and macro-benchmarks. The micro-benchmarks are data structures including Linked List, Skip List, Red-black Tree, and Hash set. The macro-benchmarks include five applications from the STAMP benchmark suite [11] (Vacation, KMeans, Genome, Labyrinth, and Intruder) and a Bank application.

For the micro-benchmarks and the bank application, we measured the transactional throughput (i.e., the number of transactions committed per second). Thus, higher is better. For the STAMP macro-benchmarks, we measured the core program execution time, which includes transactional execution time. Thus, smaller is better.

Each experiment was repeated 10 times, and each time, the VM was "warmed up" (i.e., we let the VM run the experiment for some time without logging the results) before taking the measurements. We show the average and the 90% confidence interval for each data point.

3.2 Micro-Benchmarks

We converted the micro-benchmark data structures from using course-grain locking to use transactions. The transactions contain all the code that was inside the critical sections in the course-grain locking version.

Each data structure is a representation of a sorted set of integers of size 256. The set elements are in the range 0 to 65536. Writes represent add and remove operations, and they keep the size of the set approximately constant during the experiment. Different ratios of writes and reads were used to measure the performance under different levels of contention. We also varied the number of threads in exponential steps (i.e., 2, 4, 8, ...), up to 48.

For each benchmark, we conducted experiments with different read/write ratios: 20% writes, 50% writes, 80% writes, and 100% writes.

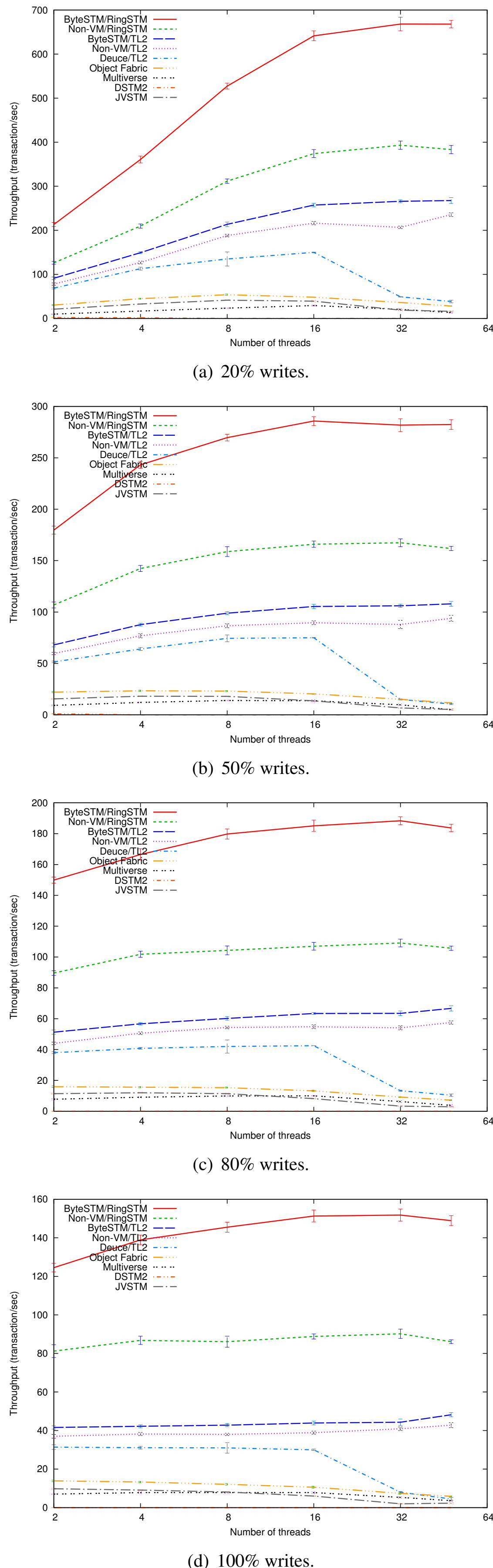
3.2.1 Linked List

Linked-list operations are characterized by a high number of reads (the range is from 70 at low contention to 270 at high contention), due to traversing the list from the head to the required node, and a few writes (about 2 only). This results in long transactions. Moreover, we observed that transactions suffer from a high number of aborts (abort ratio is from 45% to 420%), since each transaction keeps all visited nodes in its read-set, and any modification to these nodes by another transaction's add or remove will abort the transaction.

Figure 4 shows the throughput at increasing number of threads. ByteSTM has two curves representing the RingSTM and TL2 algorithms.

We observe that, in all cases, ByteSTM/RingSTM achieves the best performance and scalability. This is followed by ByteSTM/TL2. Deuce's performance degrades quickly as the number of threads is increased due to its write-set implementation. Other STMs perform in a similar way, and all of them have a very low throughput. DSTM2 has a very poor performance (not surprisingly, as it is a first generation STM implementation).

At high contention, TL2's scalability degrades. RingSTM continues to scale well up to 100% write ratio. ByteSTM/TL2 outperforms non-VM/TL2 by as much as 15% and up to 18%. ByteSTM/RingSTM outperforms non-VM/RingSTM in the 65%–71% range. The large gap between



(a) 20% writes.

(b) 50% writes.

(c) 80% writes.

(d) 100% writes.

Figure 4. Throughput under Linked List.

TL2 and RingSTM is due to the elimination of the read-set and usage of signatures in RingSTM, given the very small number of writes.

Note that the throughput *range* that we observe is lower than the throughput range reported for Java (non-VM) STM implementations such as LSA-STM and Deuce in [25, 35]. This discrepancy is directly due to our usage of the Jikes Baseline compiler and mark-and-sweep GC, which results in a slow VM. The difference in the throughput range is orthogonal to our point: ByteSTM's throughput is better than the others. With the Jikes Optimizing compiler and a high performance GC, we expect the throughput range to be consistent with, or better than those in [25, 35].

3.2.2 Skip List

Skip List operations are characterized by a medium number of reads (from 20 to 40), and a small number of writes (from 2 to 8). This results in medium-length transactions. Moreover, transactions suffer from a low number of aborts (abort ratio is from 4% to 20%).

Figure 5 shows the results. In all cases, ByteSTM/TL2 achieves the best scalability. ByteSTM/RingSTM's scalability is affected by the higher abort ratio due to Bloom filter's false positives. Among other STMs, Deuce/TL2 is the best in performance and scalability up to 16 threads. Other STMs' performance and scalability are poor. DSTM2 shows very poor performance.

ByteSTM/TL2 outperforms non-VM/TL2 in the 13–15% range. ByteSTM/RingSTM outperforms non-VM/RingSTM in the 11–16% range.

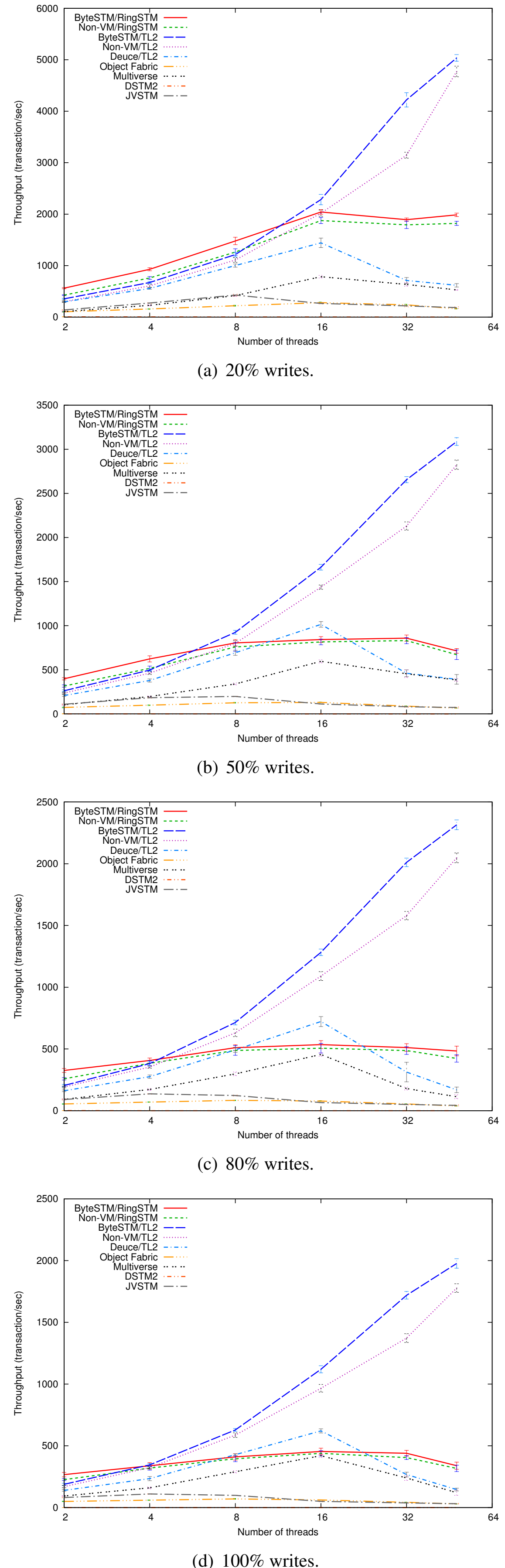
Since Deuce/TL2 achieved the best performance among all other STMs, for all further experiments, we use Deuce as a fair competitor against ByteSTM to avoid clutter, along with the non-VM implementations of TL2 and RingSTM algorithms.

3.2.3 Red-Black Tree

Red-Black Tree operations are characterized by a small number of reads (from 15 to 30), and a small number of writes (from 2 to 9). This results in short transactions. Moreover, transactions suffer from a low number of aborts (abort ratio is from 4% to 30%).

Figure 6 shows the results. We observe that, in all cases, ByteSTM/RingSTM achieves the best performance for up to 32 threads. RingSTM's performance begins to degrade after 16 threads, and with increased number of writes. This is due to the increased false positive ratio of the Bloom filter that increases the number of aborts. ByteSTM/TL2 achieves the next best performance and the best in scalability. Deuce/TL2 is the third in performance, but also suffers from performance degradation after 16 threads.

ByteSTM/TL2 outperforms non-VM/TL2 in the 13–16% range. ByteSTM/RingSTM outperforms non-VM/RingSTM in the 35–36% range. The gap between TL2 and RingSTM is not large here. This is because, the number of reads is small,



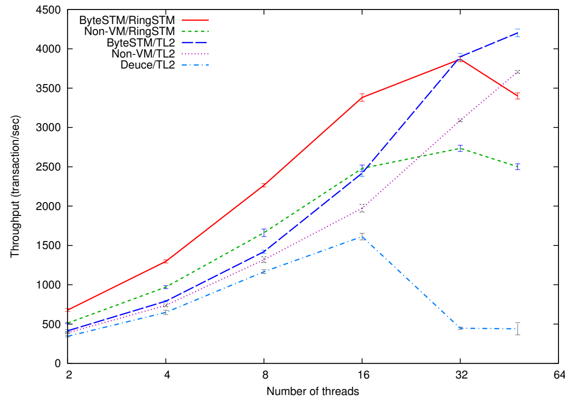
(a) 20% writes.

(b) 50% writes.

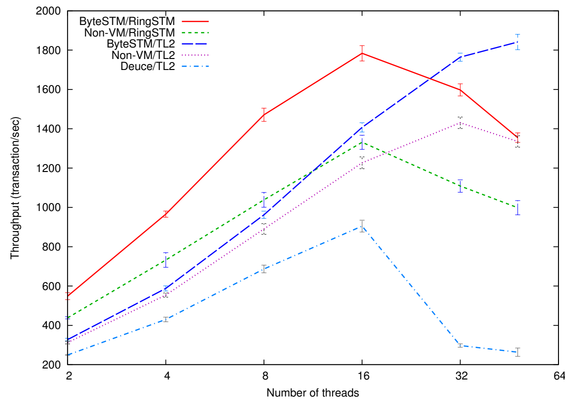
(c) 80% writes.

(d) 100% writes.

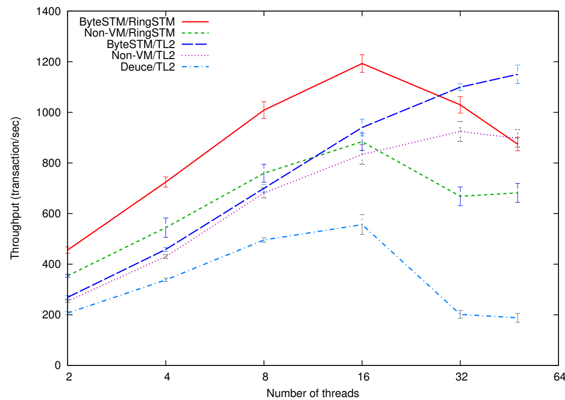
Figure 5. Throughput under Skip List.



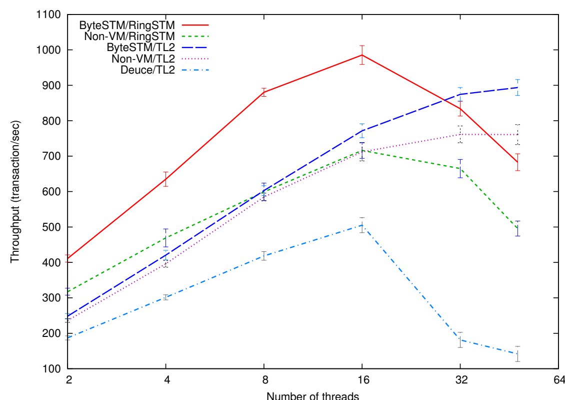
(a) 20% writes.



(b) 50% writes.



(c) 80% writes.



(d) 100% writes.

Figure 6. Throughput under Red-Black Tree.

and the number of writes is larger than that of the Linked List.

3.2.4 Hash Set

Hash Set operations are characterized by a small number of reads (from 2 to 31), and a medium number of writes (from 7 to 15). This results in short transactions. Moreover, the transactions suffer from a high number of aborts (abort ratio is from 63% to 556%) due to collisions, linked-list chains, and duplicate inserts that update the memory. The high abort ratio in this benchmark affects all implementations.

Figure 7 shows the results. ByteSTM/RingSTM achieves the best performance and scalability, followed by ByteSTM/TL2, and then Deuce. The high ratio of aborts and relatively high number of writes significantly affect Deuce's performance. Deuce does not scale well.

ByteSTM/TL2 outperforms non-VM/TL2 in the 15–22% range. ByteSTM/RingSTM outperforms non-VM/RingSTM in the 25–29% range. In this experiment, the gap between TL2 and RingSTM is not very large. This is because, the number of reads is small and the abort ratio is high.

3.3 Macro Benchmarks

3.3.1 Bank

This benchmark simulates a subset of banking operations. The benchmark is initialized with a set of accounts and an initial deposit in each account. The following operations can be done on each account: check the balance, transfer money from one account to another, and add an interest.

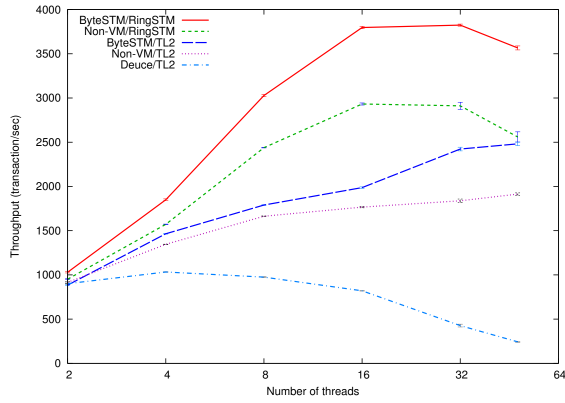
We used 8 accounts. Each thread checks the balance of all accounts, adds an interest to all the accounts, or transfers money from one random account to another random one. So, the operations are characterized by a medium number of reads (from 26 to 42), and a medium number of writes (from 11 to 22). This results in medium-length transactions. Moreover, transactions suffer a high number of aborts (abort ratio is from 188% to 268%), due to small number of accounts.

Figure 8 shows the results. In all cases, ByteSTM/RingSTM achieves the best performance, followed by ByteSTM/TL2, and then Deuce. The benchmark does not scale well for all STMs, due to high aborts.

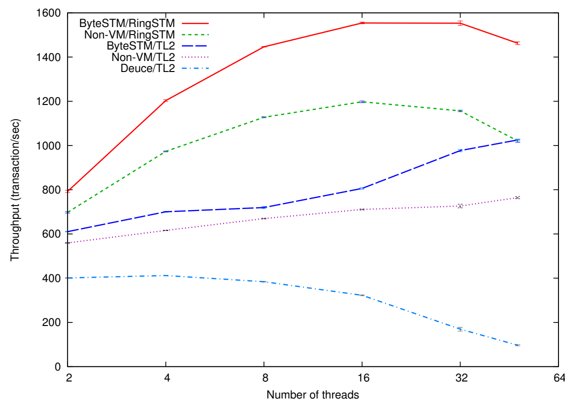
ByteSTM/TL2 outperforms non-VM/TL2 in the 10–12% range. ByteSTM/RingSTM outperforms non-VM/RingSTM in the 45–62% range.

3.3.2 Vacation

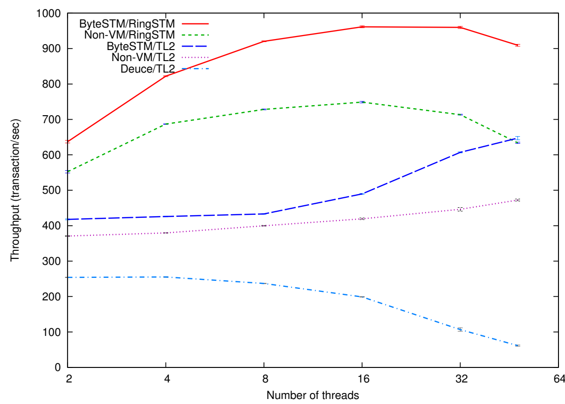
The Vacation benchmark [11] is characterized by medium-length transactions, medium read-sets, medium write-sets, and long transaction times (compared with other STAMP benchmarks). We conducted two experiments: one with low contention (a small number of operations per session, a small ratio of create/destroy operations, and operations are performed on a smaller portion of the in-memory database), and the other with high contention.



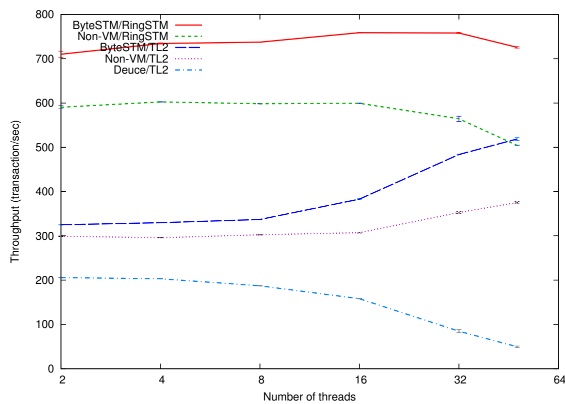
(a) 20% writes.



(b) 50% writes.

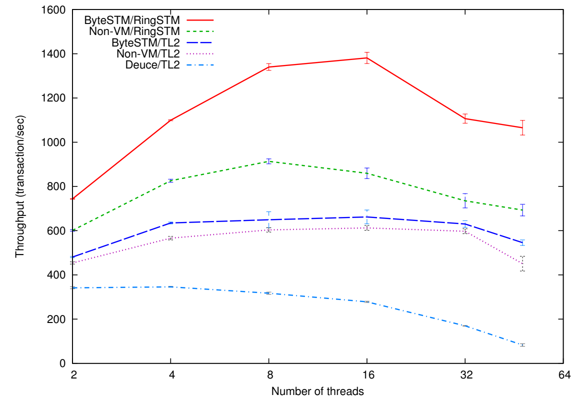


(c) 80% writes.

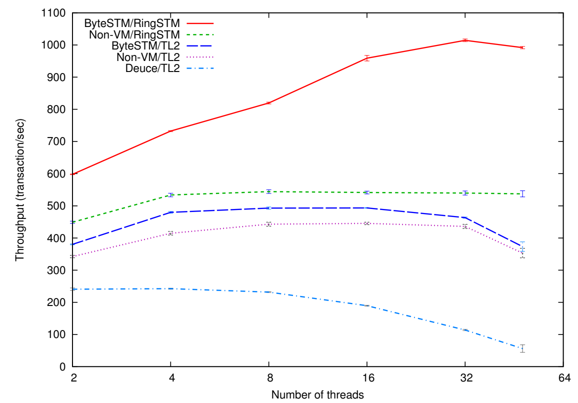


(d) 100% writes.

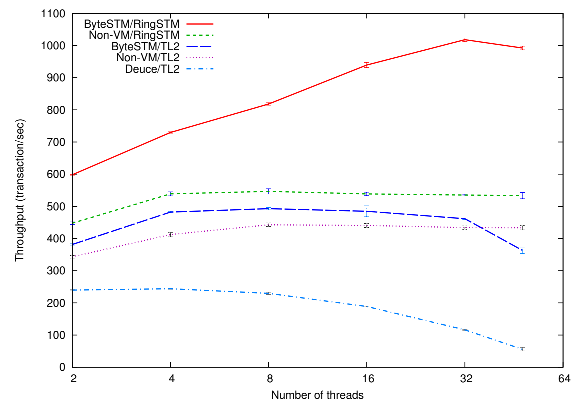
Figure 7. Throughput under Hash Set.



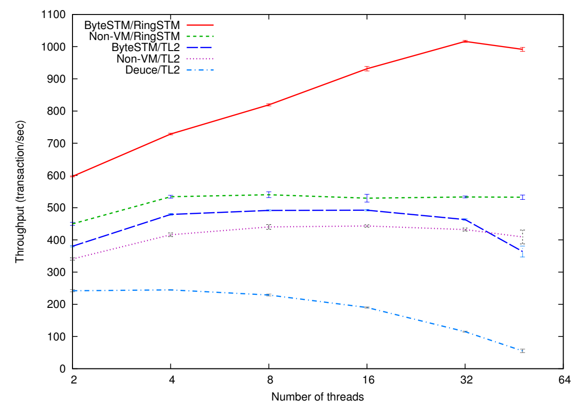
(a) 20% writes.



(b) 50% writes.

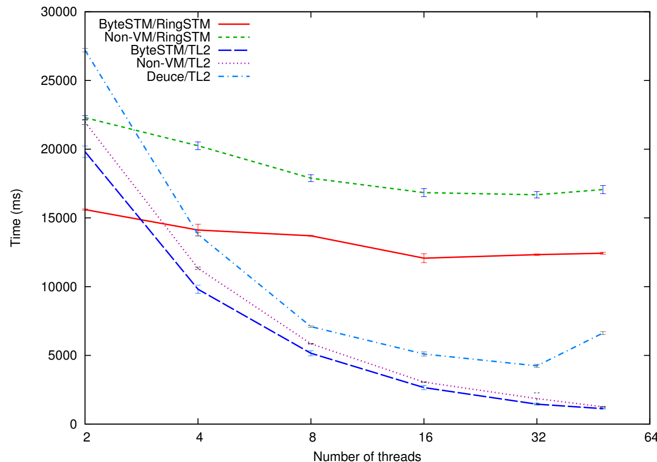


(c) 80% writes.

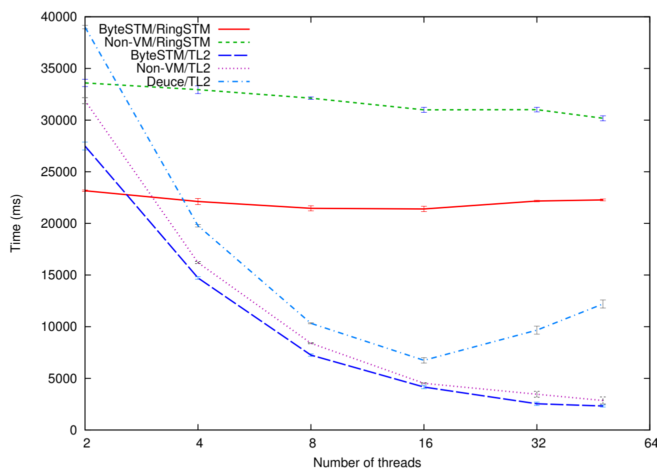


(d) 100% writes.

Figure 8. Throughput under Bank application.

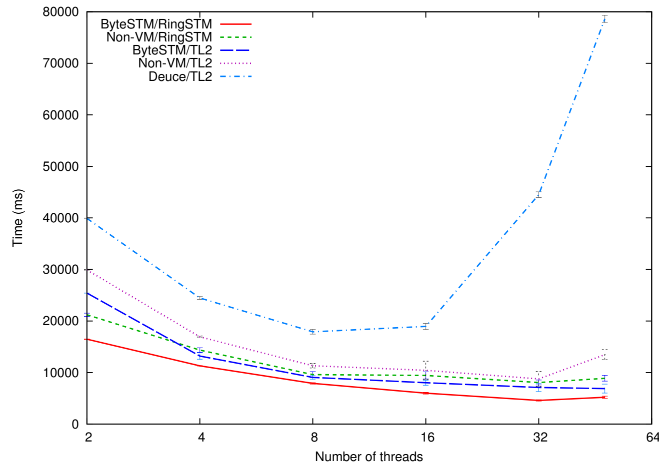


(a) Low Contention

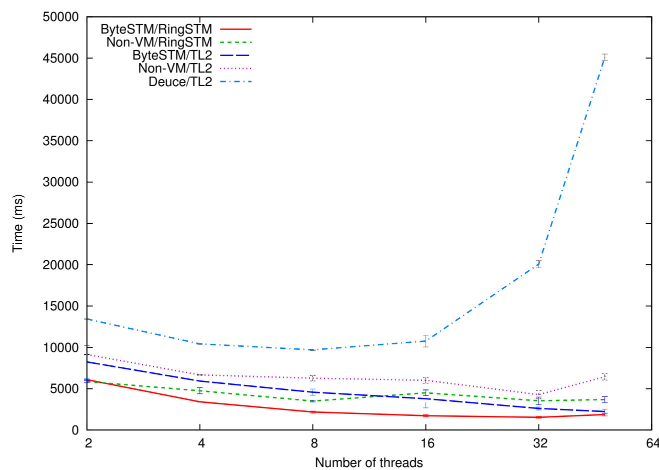


(b) High Contention

Figure 9. Execution time under Vacation.



(a) Low Contention



(b) High Contention

Figure 10. Execution time under KMeans.

Figure 9 shows the results. Note that, here, the y-axis represents the time taken to complete the experiment, and the x-axis represents the number of threads. We observe that ByteSTM/TL2 has the best performance and scalability under both low and high contention conditions. ByteSTM/RingSTM suffers from extremely high number of aborts due to false positives and long transactions. ByteSTM/TL2 outperforms non-VM/TL2 by an average of 15.7% in low contention and 18.3% in high contention. ByteSTM/RingSTM outperforms non-VM/RingSTM by an average of 38.1% in low contention and 43.9% in high contention.

3.3.3 KMeans

The KMeans benchmark [11] is characterized by short transaction lengths, small read-sets, small write-sets, and short transaction times. We conducted two experiments: one with low contention, and the other with high contention.

Figure 10 shows the results. We observe that ByteSTM with TL2 and RingSTM scales well in both cases and performs similar. ByteSTM/TL2 outperforms non-VM/TL2 by an average of 24.8% in low contention and 29.8% in high contention. ByteSTM/RingSTM outperforms non-VM/RingSTM by an average of 46.8% in low contention and 80.4% in high contention.

3.3.4 Genome

The Genome benchmark [11] is characterized by medium transaction lengths, medium read-sets, medium write-sets, long transaction times, and low contention. These characteristics are similar to that of Vacation with low contention.

Figure 11 shows the results. We see that ByteSTM/TL2 and Deuce performs the same. This graph is limited to 16 threads because the benchmark does not work with higher number of threads. Thus, the scalability of the STMs is not clear. ByteSTM/RingSTM suffers from extremely higher number of aborts due to false positives and long transactions, which is similar to the behavior observed on Vacation.

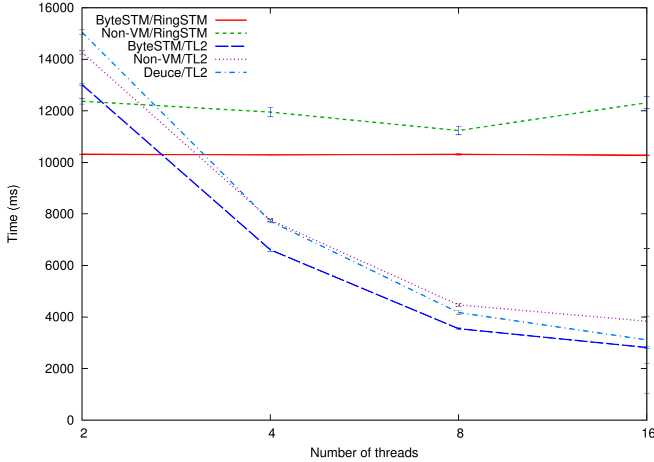


Figure 11. Execution time under Genome.

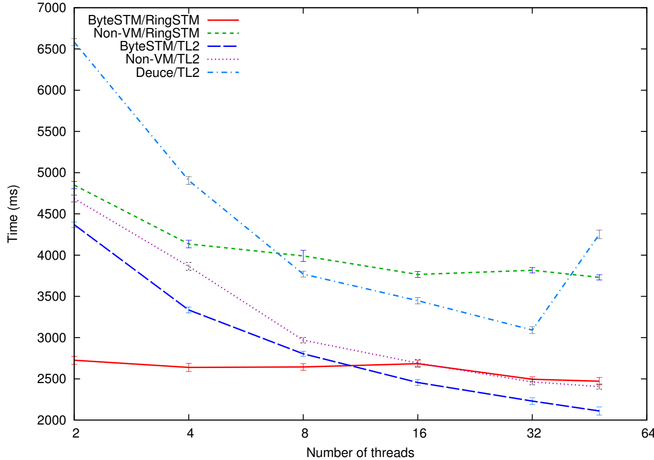


Figure 12. Execution time under Labyrinth.

3.3.5 Labyrinth

The Labyrinth benchmark [11] is characterized by long transaction lengths, large read-sets, large write-sets, long transaction times, and very high contention.

Figure 12 shows the results. We observe that ByteSTM/TL2 achieves the best performance and scalability after 16 threads. ByteSTM/RingSTM suffers from extremely high number of aborts due to false positives and long transactions, and shows no scalability. However, the high contention nature of this benchmark (i.e., all STMs suffer from high abort ratio) compensates for the false positive effect and reduces the gap between TL2 and RingSTM. ByteSTM/TL2 outperforms non-VM/TL2 by an average of 10.5%. ByteSTM/RingSTM outperforms non-VM/RingSTM by an average of 55%.

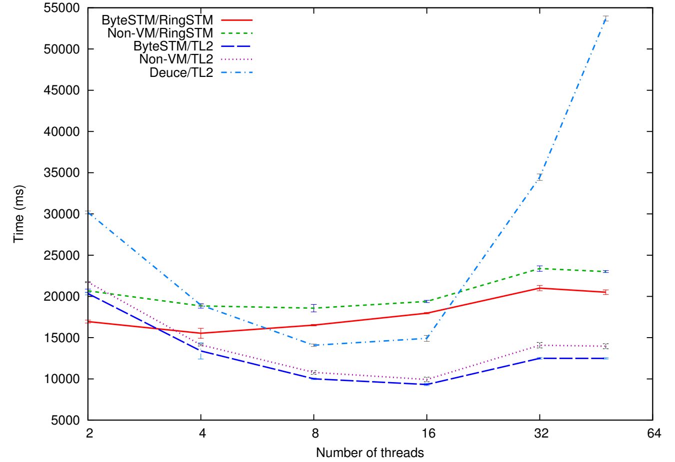


Figure 13. Execution time under Intruder.

3.3.6 Intruder

The Intruder benchmark [11] is characterized by short transaction lengths, medium read-sets, medium write-sets, medium transaction times, and high contention.

Figure 13 shows the results. We observe that ByteSTM/TL2 achieves the best performance. ByteSTM/RingSTM suffers from increased aborts due to false positives. ByteSTM/TL2 outperforms non-VM/TL2 by an average of 9%. ByteSTM/RingSTM outperforms non-VM/RingSTM by an average of 14.5%.

3.4 Summary

ByteSTM improves performance over non-VM implementations by an overall average of 30%. On micro-benchmarks, ByteSTM improves by 13% to 70%. On macro-benchmarks, ByteSTM's improvement ranges from 10% to 60%. Moreover, the scalability is significantly better. ByteSTM, in general, is better when the abort ratio is high and under high contention conditions.

RingSTM performs well, irrespective of the number of reads. However, its performance is highly sensitive to false positives when the number of writes increases. TL2 performs well when the number of reads is not large. It also performs and scales well when the number of writes increases.

4. Related Work

4.1 Library-based Implementations

Deuce [25] STM is implemented as a Java library and requires no changes to the JVM or the Java language. It implements the TL2 [15] and LSA [35] algorithms, and supports plug-ins of other STM algorithms or implementations. Deuce supports transactions implicitly (i.e., using the `@Atomic` annotation), but atomic blocks are restricted to methods. It uses instrumentation, at the bytecode level, to generate transactional code for annotated methods, which allows it to support external libraries. Deuce uses field-based

granularity, and uses `sun.misc.Unsafe` [41] non-standard proprietary API to access memory.

JVSTM [10] is a Java STM library that uses a multi-version STM algorithm. JVSTM works only on transactional objects. External libraries are not supported. Similar to Deuce, the `@Atomic` annotation is used, and offline instrumentation generates transactional code. JVSTM supports explicit transactions and uses object-based granularity.

ObjectFabric [33] is a cross-platform library, which contains an STM implementation based on XSTM [31]. XSTM is a multi-version STM and supports strong atomicity by allowing only transactional objects inside a transaction, and non-transactional code can only access the objects through (small) transactions. All shared data are immutable, and mutable data are only available privately. This way, no synchronization is required from transaction's start to commit. ObjectFabric uses explicit transactions, which only works on transactional objects, and does not support external libraries.

AtomJava [22] supports implicit transactions through atomic blocks by adding the `atomic` keyword to the Java language. To compile the new code, a source-to-source conversion is required (using Polyglot [32]), which can then be compiled using standard Java compiler. (In our experimental studies in Section 3, however, the source-to-source conversion did not work as expected.) During this conversion, transactions are used to replace atomic blocks, and objects are modified to support transactional access. Strong atomicity is supported during the code conversion. AtomJava uses object-based locking, and supports external libraries.

DSTM2 [21] is an object-based STM library. Transactional objects are created using special *factory* classes. A transactional object class is defined as a Java `interface` having the `@atomic` annotation. A DSTM2 factory object is created for each transactional object. When the factory object is created, a synthetic anonymous class is created that implements the given interface. Reflection and the bytecode engineering library [5] are used to create the class. DSTM2 uses explicit transactions that work with only transactional objects, and therefore does not support external libraries.

Multiverse [44] is a language-independent STM implementation that can work with any language running on a JVM (e.g., Scala, JRuby). Thus, it does not depend on instrumentation and uses explicit transactions. Note that, supporting only JVM languages is not really language independent. Moreover, explicit transactions are generally difficult to use and are not programmer-transparent. Multiverse uses GammaSTM [44], which is an optimized version of TL2 [15].

LSA-STM [34, 36] is an STM library that uses the LSA algorithm [35], which is a multi-version obstruction-free STM. It uses online instrumentation to generate transactional code using the ASM library [7]. Instrumentation relies on annotations. Transactional objects are marked with `@Transactional`, and atomic methods are marked with `@Atomic`. Transactions can only access transactional objects, and therefore does not support external libraries.

4.2 VM-level Implementations

Harris and Fraser modified the Sun JVM for Research (written in C) to support TM operations [18]. It supports atomic blocks using the new `atomic` keyword, and a modified compiler is used to compile the blocks into bytecode. To simplify the conversion, only methods can be atomic. Atomic methods are recognized by the VM by adding a suffix to their names during compilation, which allows supporting transactions without adding new bytecode. This C-based STM is implemented in the JVM to obtain benefits from its managed environment. The algorithm used allows inconsistent reads to occur, which can cause unrecoverable errors. For example, an inconsistent read can cause a loop to write outside the bounds of an array. In an unmanaged environment, the memory will be overwritten.

Atomos [12] presents a new programming language that is based on Java. It replaces Java monitors with transactions. The keyword `synchronized` is replaced with `atomic`, and the `wait/notify` conditional variable is replaced with `watch/retry`. Removing Java's standard synchronization mechanisms means that there is no backward compatibility for legacy code. Also, there are situations where some transactions may require different handling, such as irrevocable transactions. Atomos is implemented inside the JVM. It supports implicit transactions, strong atomicity (using an HTM that supports it), and open nesting. The JikesRVM [4] and the TCC HTM [28] are used in Atomos implementation.

Transactional monitors are added to Java in [45]. It is implemented in JikesRVM [4]. Standard Java synchronization is not removed, but it cannot be used with transactional monitors. The monitors are implemented as implicit transactions, and uses exceptions handling for transaction re-execution. The bytecode engineering library [5] is used to inject exception handling code that restores program state and restart transactions. The monitors use write and read barriers, which are functions called by JikesRVM with every read or write.

Adl-Tabatabai et. al. [1] optimize STM operations by using a JIT compiler in the ORP [13] managed environment with a new language extension. The StartJIT Java compiler [2], which is part of the VM, is modified to optimize STM calls, which are handled by McRT-STM [37]. The source code is first compiled by Polyglot [32] to handle the new language extensions. The JIT compiler then optimizes the STM calls that use McRT-STM.

In contrast to these STMs, ByteSTM works with all data types, not just transactional objects. It supports external libraries, does not use instrumentation, and uses field-based granularity. ByteSTM directly accesses memory without using non-standard libraries, supports atomic blocks anywhere in the code, and its metadata bypasses the GC. It supports implicit transactions, works with or without compiler sup-

port, works at the VM-level, provides opacity, and is implemented entirely in software.

However, ByteSTM (currently) has no multi-version STM algorithm implementation, and does not support strong atomicity, nesting, or conditional variables. Also, ByteSTM (currently) does not support plug-ins. Besides the directions in Section 2.7, these are also planned as future work.

As previously described, Table 1 summarizes our comparison of ByteSTM with competitor STM implementations.

5. Conclusions

At its core, our work shows that implementing an STM at the VM-level is indeed possible, and can yield significant performance benefits. This is because, at the VM-level, STM overhead is significantly reduced. Additionally, memory operations are faster, the GC overhead is eliminated, no instrumentation is required (i.e., the same code can run in two modes: transactional and non-transactional). Moreover, atomic blocks can be supported anywhere, and metadata is attached to the thread header. Since the VM has full control over all transactional and non-transactional memory operations, features such as strong atomicity and support for irrevocable operations (which are not currently supported) can be efficiently supported.

These optimizations are not possible at a library-level. Moreover, compiler-level STM for VM languages cannot support these optimizations also. Thus, implementing an STM for a managed language at the VM-level is likely the most performant.

ByteSTM is publicly available at `hydravm.org/bytestm`.

References

[1] A. Adl-Tabatabai, B. Lewis, et al. Compiler and runtime support for efficient software transactional memory. *ACM SIGPLAN Notices*, 41(6):26–37, 2006.

[2] A. Adl-Tabatabai et al. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 2003.

[3] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *PPoPP*, pages 163–174. ACM, 2008. ISBN 978-1-59593-795-7.

[4] B. Alpern, S. Augart, et al. The Jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44:399–417, January 2005. ISSN 0018-8670.

[5] Apache Software Foundation. BCEL: Byte-code engineering library. `http://jakarta.apache.org/bcel/manual.html`, 2011.

[6] J. a. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapalka. Leveraging parallel nesting in transactional memory. In *PPoPP*, pages 91–100. ACM, 2010. ISBN 978-1-60558-877-3.

[7] W. Binder, J. Hulaas, and P. Moret. Advanced Java bytecode instrumentation. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 135–144. ACM, 2007.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970. ISSN 0001-0782.

[9] B. J. Bradel and T. S. Abdelrahman. The use of hardware transactional memory for the trace-based parallelization of recursive Java programs. In *PPPJ*, pages 101–110, 2009.

[10] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

[11] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, September 2008.

[12] B. Carlstrom, A. McDonald, et al. The Atomos transactional programming language. *ACM SIGPLAN Notices*, 41(6):1–13, 2006.

[13] M. Cierniak et al. The open runtime platform: A flexible high-performance managed runtime environment. *Intel Technology Journal*, 2003.

[14] I. Corporation. Intel C++ STM Compiler 4.0, Prototype Edition. `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`, 2009.

[15] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *DISC*, 2006.

[16] J. Gottschlich and D. Connors. DracoSTM: A practical C++ approach to software transactional memory. In *Symposium on Library-Centric Software Design*, pages 52–66. ACM, 2007.

[17] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, pages 175–184. ACM, 2008.

[18] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, 2003.

[19] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. *ACM SIGARCH computer architecture news*, 21(2):289–300, 1993.

[20] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[21] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *ACM SIGPLAN Notices*, 41(10):253–262, 2006.

[22] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *Workshop on Memory system performance and correctness*, pages 82–91. ACM, 2006.

[23] R. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Sept. 1996. ISBN 0471941484.

[24] G. Kestor, S. Stipic, et al. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In *TRANSACT*, 2009.

[25] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *MULTIPROG*, 2010.

[26] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, MIT, 2004.

[27] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM*, 3(4): 184–195, Apr. 1960. ISSN 0001-0782.

[28] A. McDonald, J. Chung, et al. Characterization of TCC on chip-multiprocessors. In *PACT*, pages 63–74. IEEE, 2005.

[29] F. Meawad, R. Macnak, and J. Vitek. Collecting transactional garbage. In *TRANSACT*, 2011.

[30] J. Moss and A. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006.

[31] C. Noël. Extensible software transactional memory. In *C* Conference on Computer Science and Software Engineering*, pages 23–34, 2010.

[32] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction*, pages 138–152. Springer, 2003.

[33] ObjectFabric Inc. ObjectFabric. `http://objectfabric.com`, 2011.

[34] T. Riegel, P. Felber, and C. Fetzer. LSA-STM. `http://tmware.org/lsastm`, 2006.

[35] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. *Distributed Computing*, pages 284–298, 2006.

[36] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. *TRANSACT*, 298, 2006.

[37] B. Saha, A. Adl-Tabatabai, et al. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, pages 187–197, 2006.

[38] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[39] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *SPAA*, pages 275–284, 2008.

[40] Sun Microsystems. Java Annotations. `http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html`, `http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html`, 2004.

[41] Sun Microsystems. `sun.misc.Unsafe`. `http://www.docjar.com/docs/api/sun/misc/Unsafe.html`, 2006.

[42] The Jikes RVM Project. Baseline compiler. `http://jikesrvm.org/Baseline+Compiler`, 2005.

[43] The Jikes RVM Project. Optimizing compiler. `http://jikesrvm.org/Optimizing+Compiler`, 2005.

[44] P. Veentjer. Multiverse. `http://multiverse.codehaus.org`, 2011.

[45] A. Welc, S. Jagannathan, and A. Hosking. Transactional monitors for concurrent objects. *ECOOP*, pages 494–514, 2004.