ByteSTM: Virtual Machine-level Java Software Transactional Memory

Mohamed Mohamedin, Binoy Ravindran, and Roberto Palmieri

ECE Dept., Virginia Tech, Blacksburg, VA, USA
{mohamedin,binoy,robertop}@vt.edu

Abstract. We present ByteSTM, a virtual machine-level Java STM implementation that is built by extending the Jikes RVM. We modify Jikes RVM's optimizing compiler to transparently support implicit transactions. Being implemented at the VM-level, it accesses memory directly, avoids Java garbage collection overhead by manually managing memory for transactional metadata, and provides pluggable support for implementing different STM algorithms to the VM. Our experimental studies reveal throughput improvement over other non-VM STMs by 6–70% on micro-benchmarks and by 7–60% on macro-benchmarks.

1 Introduction

Transactional Memory (TM) [13] is an attractive programming model for multicore architectures promising to help the programmer in the implementation of parallel and concurrent applications. The developer can focus the effort on the implementation of the application's business logic, giving the responsibility to managing concurrency to the TM framework. It ensures, in a completely transparent manner, properties difficult to implement manually like atomicity, consistency, deadlock-freedom and livelock-freedom. The programmer simply organizes the code identifying blocks to be executed as transactions (so called atomic blocks). TM overtakes the classical coarse grain lock-based implementation of concurrency, executing transactions optimistically and logging into a private part of the memory the results of read and write operations performed during the execution (respectively on the read- and write-set). Further TM is composable.

TM has been proposed in hardware (HTM; e.g., [8]), in software (STM; e.g., [17]), and in combination (HybridTM; e.g., [18]). 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.

Given STM's hardware-independence, 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 classified into those that use *explicit* transactions [14,21,26] and those that use *implicit* transactions [5, 17, 23]. Explicit transactions are difficult to use. They

support only transactional objects (i.e., objects that are known to the STM library by implementing a specific interface, being annotated, etc) and hence cannot work with external libraries. Implicit transactions, on the other hand, use modern language features (e.g., Java annotations) to mark code sections as `atomic`. Instrumentation is used to transparently add transactional code to the atomic sections (e.g., begin, transactional reads/writes, commit). Some implicit transactions work only with transactional objects [5, 23], while others work on any object and support external libraries [17].

Compiler-based STMs (e.g., [15, 16]) 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 external libraries' source code to instrument it and add transactional support. Usually, external libraries source code is not available. 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 [1,7,12,27]. [12] is implemented in C inside the JVM to get benefits of the VM-managed environment, and uses an algorithm that does not ensure the opacity correctness property [11]. This means that inconsistent reads may occur before a transaction is aborted, causing unrecoverable errors in an unmanaged environment. Thus, the VM-level implementation choice is to prevent such unrecoverable errors, which are not allowed in a managed environment. [7] presented a new Java-based programming language called Atomos, and a concomitant VM where standard Java synchronization (i.e., `synchronized`, `wait/notify`) is replaced with transactions. However, in this work, transactional support relies 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, the Deuce STM library [17] uses a non-standard proprietary API (i.e., `sun.misc.Unsafe`) for performance gains, which is 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, which allows faster write-backs to memory. The VM also has full control of the GC, which allows minimizing the GC's degrading effect on STM performance (see Subsection 2.5). Moreover, if TM is supported using hardware (as in [7]), then VM is the only appropriate level of abstraction to exploit that support for obtaining higher performance. (Otherwise, if TM is supported at a higher

level, the GC will abort transactions when it interrupts them.) Also, VM memory systems typically use a centralized data structure, which increases conflicts, degrading performance [4].

Table 1. Comparison of Java STM implementations.

Feature	Deuce [17]	JVSTM [5]	ObjectFabric [21]	AtomJava [15]	DSTM2 [14]	Multiverse [26]	LSA-STM [23]	Harris & Fraser [12]	Atomos[1] [7]	Trans. monitors [27]	ByteSTM
Implicit transactions	√	√	X	√	X	X	√	√	√	√	√
All data types	√	X	X	√	X	X	X	√	√	√	√
External libraries	√	X	X	√ [2]	X	X	X	√	X [3]	√	√
Unrestricted atomic blocks	X	X	√	√	√	√	X	√	√	√	√
Direct memory access	√[4]	X	X	X	X	X	X	√	√	X	√
Field-based granularity	√	X	X	X	X	X	X	X	X	X	√
No GC overhead	√ [5]	X	X	X	X	X	X	√	√	X	√
Compiler support	X	X	X	√	X	X	X	√	√	√	√ & X [6]
Strong atomicity	X	X	√	√	X	X	X	X	√	X	X
Closed/Open nesting	X	√	√	X	X	X	X	X	√	X	X
Conditional variables	X	X	X	X	X	X	X	X	√	X	X

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

Motivated by these observations, we design and implement a VM-level STM: ByteSTM. In ByteSTM, a transaction can surround any block of code, and it is not restricted to methods. Memory bytecode instructions reachable from a transaction are translated so that the resulting native code executes transactionally. ByteSTM uniformly handles all data types (not just transactional objects), using the memory address and number of bytes as an abstraction, and thereby supports external libraries. It uses field-based granularity, which scales better than object-based or word-based granularity, and eliminates the GC overhead, by manually managing (e.g., allocation, deallocation) memory for transactional metadata. It can work without compiler support, which is only required if the new language construct `atomic` is used. ByteSTM has a modular architecture, which allows different STM algorithms to be easily plugged in (we have implemented three algorithms: TL2 [10], RingSTM [25], and NOrec [9]). Table 1 distinguishes ByteSTM from other STM implementations. The current release of

ByteSTM does not support closed/open nesting, strong atomicity, or conditional variables. However, features like these are in the implementation roadmap.

ByteSTM is open-sourced and is freely available at `hydravm.org/bytestm`.

2 Design and Implementation

ByteSTM is built by modifying Jikes RVM [2] using the optimizing compiler. Jikes RVM is a Java research virtual machine and it is implemented in Java. Jikes RVM has two types of compilers: the Optimizing compiler and the Baseline compiler. The Baseline compiler simulates the Java stack machine and has no optimization. The Optimizing compiler does several optimizations (e.g., register allocation, inlining, code reordering). Jikes RVM has no interpreter, and bytecode must be compiled to native code before execution. Building the Jikes RVM with production configuration gives performance comparable to the HotSpot server JIT compiler [22].

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

ByteSTM is implicitly transactional: the program only specifies the start and end of a 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 the transactional mode. In this mode, all writes are isolated and the execution of the instructions proceeds optimistically 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, only at this point, all transaction modifications become externally visible to other transactions. If the commit fails, all the modifications are discarded and the transaction restarts from the beginning.

We modified the Jikes optimizing compiler. Each memory load/store instruction (`getfield`, `putfield`, `getstatic`, `putstatic`, and all array access instructions) is replaced with a call to a corresponding method that adds the transactional behavior to it. The compiler inlines these methods to eliminate the overhead of calling a method with each memory load/store. The resulting behavior is that each instruction checks whether the thread is running in transactional or non-transactional mode. Thus, instruction execution continues transactionally or non-transactionally. The technique is used to translate the

new instructions `xBegin` and `xCommit` (or replacing calls to `stm.STM.xBegin()` and `stm.STM.xCommit()` with the correct method calls).

Modern STMs [5, 17, 23] 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 transactions. This conversion does not need the source code and can be done offline or online. Instrumentation allows using external libraries – i.e., code inside a transaction can call methods from an external library, which may modify program data [17].

In ByteSTM, 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 by first checking the thread's mode. If the mode is transactional, the thread runs transactionally; otherwise, it runs regularly. 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 (excluding blocks containing irrevocable operations such as I/O). 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 [17].

2.1 Metadata

Working at the VM level allows changing the thread header without modifying program code. For each thread that executes transactions, the metadata added includes the read-set, the write-set, and other STM algorithm-specific metadata. Metadata is added to the thread header and is used by all transactions executed in the thread. 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.

2.2 Memory Model

At the VM-level, the physical memory address of each object's field can be easily obtained. Since ByteSTM 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. Since arrays are objects in Java, memory accesses to arrays are tracked at the element level, which eliminates unnecessary aborts.

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). Thus, our memory model is simplified as: base object plus an offset for all cases.

Using absolute addresses is limited to non-moving GC only (i.e., a GC which releases unreachable objects without moving reachable objects, like the mark-and-sweep GC). In order to support moving GC, a field is represented by its base object and the field's offset within that object. When the GC moves an object, only the base object's address is changed. All offsets remain the same. ByteSTM's write-set is part of the GC root-set. Thus, the GC automatically changes the saved base objects' addresses as part of its reference updating phase.

To simplify how the read-set and the write-set are handled, we use a unified memory access scheme. At a memory load, the information needed to track the read includes the base object and the offset within that object of the read field. At a memory store, the base object, the field's offset, 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 (base object, offset, 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 the same, 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 $O(n)$. To obtain the benefits of arrays and hashing's speed, open-addressing hashing with linear probing is used. We used an array of size 2^n, which simplifies the modulus calculation.

We used Java's `System.identityHashCode` standard method and configured Jikes to use the memory address to compute an object's hash code. This method also handles object moving. We then add the field's offset to the returned hash code, and finally remove the upper bits from the result using bitwise *and* operation (which is equivalent to calculating the modulus): *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 efficient with addresses, as the collision ratio is small. Moreover, as program variables are aligned in memory, when a collision happens, there is always an empty cell

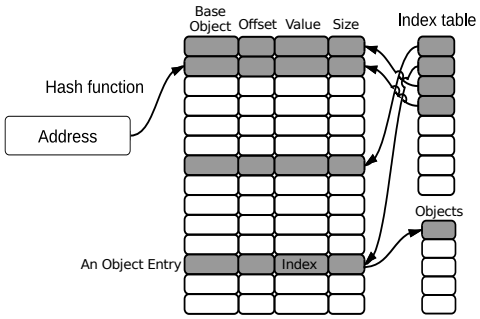

Fig. 1. ByteSTM's write-set using open address hashing.

after the required index because of the memory alignment gap (so linear probing will give good results). This way, we have a fast and efficient hashing function that adds 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.

Open addressing has two drawbacks: memory overhead and rehashing. These can be mitigated by choosing the array size such that the number of rehashing is reduced, while minimizing memory usage. Figure 1 shows how ByteSTM's write-set is represented using open-addressing. In this figure, the field's memory address is hashed to determine the associated index in the array. Four entries are occupied in the shown array (identified by gray color) and they are scattered by the hashing function into different locations in the array. The index table contains all used indices in the main array contiguously for faster sequential access to the write-set entries. Reference fields (object entries) are handled differently as described in Subsection 2.5.

2.4 Atomic Blocks

ByteSTM supports atomic blocks anywhere in the code, excluding I/O operations and JNI native calls. When `xBegin` is executed, local variables are backed up. If a transaction is aborted, the variables are restored and the transaction can restart as if nothing has changed in the variables. This technique simplifies the handling of local variables since there is no need to monitor them.

ByteSTM has been designed to natively support opacity [11]. In fact, when an inconsistent read is detected in a transaction, the transaction is immediately aborted. Local variables are then restored, and the transaction is restarted by throwing an exception. The exception is caught just before the end of the transaction loop so that the loop continues again. Note that throwing an exception is not expensive if the exception object is preallocated. Preallocating the exception object eliminates the overhead of creating the stack trace every time the

exception is thrown. Stack trace is not required for this exception object since it is used only for doing a long jump. The result is similar to `setjmp/longjmp` in C.

2.5 Garbage Collector

One major drawback of building an STM for Java (or any managed language) is the GC [19]. 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 STM overhead.

Some STMs solve this problem by reducing memory allocation and recycling allocated memory. For example, [17] uses object pooling, 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 checks 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 referenced by the program's variables. 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 therefore be writing a dangling pointer. We solve this problem by modifying the behavior of adding an object to the write-set. Instead of storing the object address in the write-set entry value, the object is added to another array (i.e., an "objects array"). The object's index in the objects array is stored in the write-set entry value (Figure 1). Specifically, if an object contains another object (e.g., a field that is a reference), we cannot save the field value as a primitive type (e.g., the absolute address) since the address can be changed by the GC. The field value is therefore saved as an object in the objects array which is available to the set of roots that the GC scans. The write-set array is another source of roots. So, the write-set contains the base objects and the objects array contains the object fields within the base objects. This prevents the GC from reclaiming the objects. Our approach is compatible with all GC available in Jikes RVM and we believe that this approach is better than modifying a specific GC.

2.6 STM Algorithms

ByteSTM's modular architecture allows STM algorithms to be easily "plugged in." We implemented three algorithms: TL2 [10], RingSTM [25], and NOrec [9]. Our rationale for selecting these three is that, they are the best performing algorithms reported in the literature. Additionally, they cover different points in the performance/workload tradeoff space: TL2 is effective for long transactions, moderate number of reads, and scales well with large number of writes. RingSTM is effective for transactions with high number of reads and small number of writes. NOrec has a better performance with small number of cores and has no false conflicts since it validates by value.

Plugging a new algorithm in ByteSTM is straight forward. One needs to implement read barriers, write barriers, transaction start, transaction end, and any other algorithm specific helping methods. All these methods are in one class "STM.java". No prior knowledge of Jikes RVM is required and porting a new algorithm to ByteSTM requires only understanding ByteSTM framework.

3 Experimental Evaluation

To understand how ByteSTM, a VM-level STM, stacks up against non VM-level STMs, we conducted an extensive experimental study. Though a VM-level implementation may help improve STM performance, the underlying STM algorithm used also plays an important role, especially, when that algorithm's performance is known to be workload-dependent (see Subsection 2.6). Thus, the performance study wants also to investigate whether any performance gain from a VM-level implementation is *algorithm-independent*. It would also be interesting to understand whether some algorithms gained more than others, and if so, why. Thus, we compare ByteSTM against non-VM STMs, with the same algorithm inside the VM versus "outside" it.

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

The non-VM implementations were made as close as possible to the VM ones. Offline instrumentation was used to eliminate online instrumentation overhead. The same open-address hashing write set was used. A large read-set and write-set were used so that they were sufficient for the experiments without requiring extra

space. The sets were recycled for the next transactions, thereby needing only a single memory allocation and thus minimizing the GC overhead. We used Deuce for the non-VM implementation, since it has many of ByteSTM's features – e.g., it directly accesses memory and uses field granularity. Moreover, it achieved the best performance among all competitors (see results later in this section).

3.1 Test Environment

We used a 48-core machine, which has four AMD Opteron^{TM}Processors, each with 12 cores running at 1700 MHz, and 16 GB RAM. The machine runs Ubuntu Linux 10.04 LTS 64-bit. We used Jikes's production configuration (version 3.1.2), which includes the Jikes optimizing compiler and the GenImmix GC [3] (i.e., a two-generation copying GC) and matches ByteSTM configurations. Since Deuce uses a non-standard proprietary API, `sun.misc.Unsafe`, which is not fully supported by Jikes, to run Deuce atop Jikes RVM, we added the necessary methods to Jikes RVM's `sun.misc.Unsafe` implementation (e.g., `getInt`, `putInt`, etc).

Our test applications include both micro-benchmarks (i.e., data structures) and macro-benchmarks. The micro-benchmarks include Linked-List, Skip-List, Red-Black Tree, and Hash Set. The macro-benchmarks include five applications from the STAMP suite [6]: Vacation, KMeans, Genome, Labyrinth, and Intruder. We used Arie Zilberstein's Java implementation of STAMP [28].

For the micro-benchmarks, we measured the transactional throughput (i.e., transactions committed per second). Thus, higher is better. For the 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 for some time without logging the results) before taking the measurements. We show the average for each data point. Due to space constraints, we only show results for Linked-List, Red-Black Tree, Vacation, Intruder, and Labyrinth (see [20] for complete results).

3.2 Micro-Benchmarks

We converted the data structures from coarse-grain locks to transactions. The transactions contain all the critical section code in the coarse-grain lock version.

Each data structure is used to implement a sorted integer set interface, with set size 256 and set elements in the range 0 to 65536. Writes represent add and remove operations, and they keep the set size approximately constant during the experiments. Different ratios of writes and reads were used to measure performance under different levels of contention: 20% and 80% writes. We also varied the number of threads in exponential steps (i.e., 1, 2, 4, 8, ...), up to 48.

Linked List. Linked-list operations are characterized by 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 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 operation will abort the transaction.

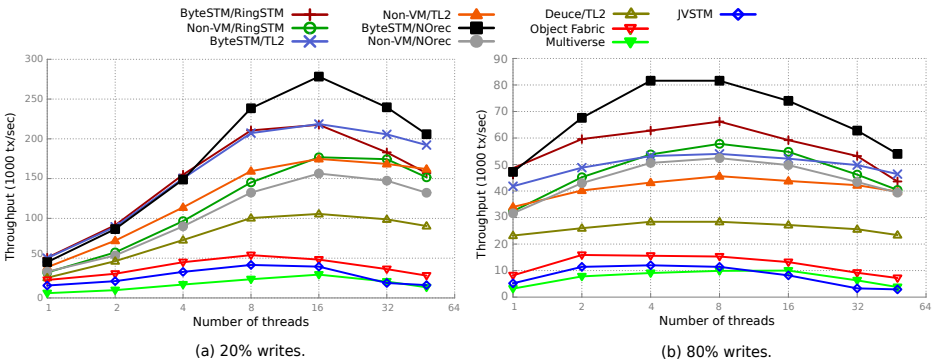

Fig. 2. Throughput for Linked-List.

Figure 2 shows the results. ByteSTM has three curves: RingSTM, TL2, and NOrec. In all cases, ByteSTM/NOrec achieves the best performance since it uses an efficient read-set data structure based on open addressing hashing. ByteSTM/RingSTM comes next since it has no read-set and it uses a bloom filter as a read signature, but the performance is affected by bloom filter's false positives. This is followed by ByteSTM/TL2 which is affected by its sequential read-set. Deuce's performance is the best between other STM libraries. Other STMs perform similarly, and all of them have very low throughput. Non-VM implementation of each algorithm performs in a similar manner but with lower throughput. ByteSTM outperforms non-VM implementations by 15–70%.

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, RingSTM and NOrec. Accordingly, in the rest of the plots the curves of JVSTM, Multiverse and Object Fabric will be dropped (simplifying also the legibility).

Red-black Tree. Here, operations are characterized by small number of reads (15 to 30), and small number of writes (2 to 9), which result in short transactions.

Figure 3 shows the results. In all cases, ByteSTM/TL2 achieves the best performance and scalability due to the small number of reads. ByteSTM/NOrec does not scale well since it is based on one single global lock. Moreover, the small size of the tree creates a high contention between the threads. RingSTM's performance is similar to NOrec as it is based on a global data structure and has the added overhead of bloom's filter false positives.

In this benchmark, there is no big gap between ByteSTM and non-VM implementation due to the nature of the benchmark, namely very short transactions

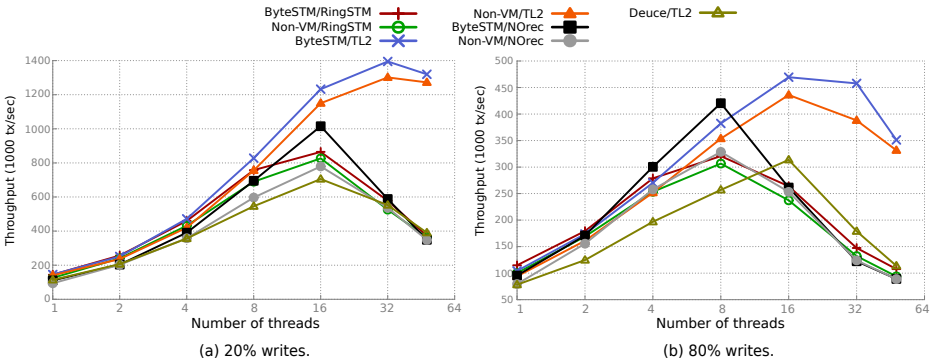

Fig. 3. Throughput of Red-Black Tree.

over a small data structure by large number of threads. ByteSTM outperforms non-VM implementation by 6–17%.

3.3 Macro Benchmark

Vacation. Vacation has medium-length transactions, medium read-sets, medium write-sets, and long transaction times (compared with other STAMP benchmarks). We conducted two experiments: low contention and high contention.

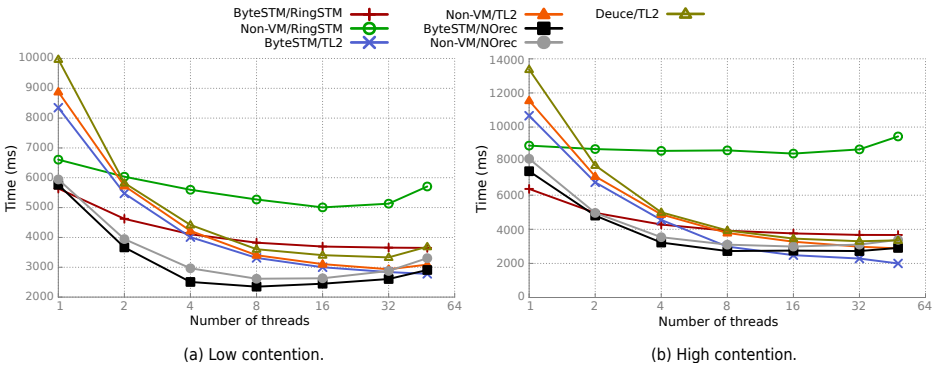

Fig. 4. Execution time under Vacation.

Figure 4 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. ByteSTM/NOrec has the best performance under both low and high contention conditions. The efficient read-set implementation contributed to its performance. But, it does not scale well. ByteSTM/RingSTM suffers from high number of aborts due to false positives and long transactions so it started with

a good performance then degrades quickly. ByteSTM outperforms non-VM implementations by an average of 15.7% in low contention and 18.3% in high contention.

Intruder. The Intruder benchmark [6] is characterized by short transaction lengths, medium read-sets, medium write-sets, medium transaction times, and high contention.

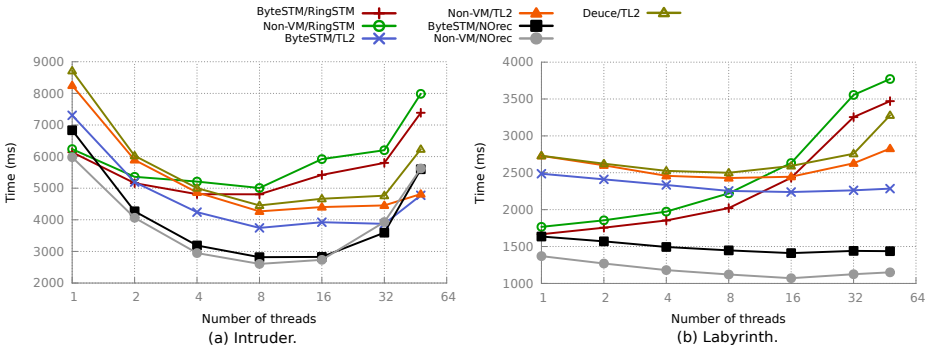

Fig. 5. Execution time under (a) Intruder, and (b) Labyrinth.

Figure 5(a) shows the results. We observe that ByteSTM/NOrec achieves the best performance but does not scale. ByteSTM/RingSTM suffers from increased aborts due to false positives. ByteSTM/TL2 has a moderate performance. ByteSTM outperforms non-VM implementations by an average of 11%.

Labyrinth. The Labyrinth benchmark [6] is characterized by long transaction lengths, large read-sets, large write-sets, long transaction times, and very high contention.

Figure 5(b) shows the results. ByteSTM/NOrec achieves the best performance. ByteSTM/RingSTM suffers from extremely high number of aborts due to false positives and long transactions, and shows no scalability. All algorithms suffered from the high contention and has very low scalability. ByteSTM outperforms non-VM implementation by an average of 18.5%.

3.4 Summary

ByteSTM improves over non-VM implementations by an overall average of 30%: on micro-benchmarks, it improves by 6 to 70%; on macro-benchmarks, it improves by 7 to 60%. Moreover, scalability is better. ByteSTM, in general, is better when the abort ratio and contention are high.

RingSTM performs well, irrespective of reads. However, its performance is highly sensitive to false positives when writes increases. TL2 performs well when reads are not large due to its sequential read-set implementation. It also performs and scales well when writes increases. NOrec performs the best with behavior

similar to RingSTM as it does not suffer from false positives, but it does not scale.

Algorithms that uses a global data structure and small number of CAS operations (like NOrec and RingSTM) have a bigger gain when implemented at VM-level. TL2 that uses a large number of locks (e.g., Lock table) and hence large number of CAS operations did not gain a lot from VM-level implementation.

4 Conclusions

Our work shows that implementing an STM at the VM-level can yield significant performance benefits. This is because at the VM-level: *i)* memory operations are faster; *ii)* the GC overhead is eliminated; *iii)* STM operations are embedded in the generated native code; *iv)* metadata is attached to the thread header.

For all these reasons the overhead of STM is minimized. Since the VM has full control over all transactional and non-transactional memory operations, features such as strong atomicity and irrevocable operations (not currently supported) can be efficiently supported.

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

Next steps for ByteSTM are exploring compile time optimization specific for STM (i.e., STM optimization pass). And, modify the thread scheduler so that it will be STM aware and reduces the conflicts rate.

ByteSTM is open-sourced and is freely available at `hydravm.org/bytestm`. A modified version of ByteSTM is currently used in the HydraVM project [24], which is exploring automated concurrency refactoring in legacy code using TM.

Acknowledgments. This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385. A short version of this paper appeared as Brief Announcement at 2013 TRANSACT workshop. TRANSACT does not have archival proceedings and explicitly encourages resubmissions to formal venues.

References

1. A. Adl-Tabatabai et al. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 2003.
2. 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.
3. S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI*, 2008.
4. B. J. Bradel and T. S. Abdelrahman. The use of hardware transactional memory for the trace-based parallelization of recursive Java programs. In *PPPJ*, 2009.

5. J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.
6. C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, September 2008.
7. B. Carlstrom, A. McDonald, et al. The Atomos transactional programming language. *ACM SIGPLAN Notices*, 41(6):1–13, 2006.
8. D. Christie, J. Chung, et al. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *EuroSys*, pages 27–40, 2010.
9. L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP*, pages 67–78. ACM, 2010.
10. D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *DISC*, 2006.
11. R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, pages 175–184, 2008.
12. T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, 2003.
13. T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.
14. M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *ACM SIGPLAN Notices*, 41(10):253–262, 2006.
15. B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *Workshop on Memory system performance and correctness*, pages 82–91, 2006.
16. Intel Corporation. Intel C++ STM Compiler, 2009. `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`.
17. G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *MULTIPROG*, 2010.
18. S. Lie. Hardware support for unbounded transactional memory. Master's thesis, MIT, 2004.
19. F. Meawad et al. Collecting transactional garbage. In *TRANSACT*, 2011.
20. M. Mohamedin and B. Ravindran. ByteSTM: Virtual Machine-level Java Software Transactional Memory. Technical report, Virginia Tech, 2012. Available at `http://www.hydravm.org/hydra/chrome/site/pub/ByteSTM_tech.pdf`.
21. ObjectFabric Inc. ObjectFabric. `http://objectfabric.com`, 2011.
22. M. Paleczny, C. Vick, and C. Click. The Java Hotspot$^{\text{TM}}$Server Compiler. In *Java^{TM} Virtual Machine Research and Technology Symposium*. USENIX, 2001.
23. T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. *TRANSACT*, 2006.
24. M. M. Saad, M. Mohamedin, and B. Ravindran. HydraVM: extracting parallelism from legacy sequential code using STM. In *HotPar*. USENIX, 2012. `hydravm.org`.
25. M. F. Spear et al. RingSTM: scalable transactions with a single atomic instruction. In *SPAA*, pages 275–284, 2008.
26. P. Veentjer. Multiverse. `http://multiverse.codehaus.org`, 2011.
27. A. Welc et al. Transactional monitors for concurrent objects. In *ECOOP*, 2004.
28. A. Zilberstein. Java implementation of STAMP. `https://github.com/DeuceSTM/DeuceSTM/tree/master/src/test/jstamp`, 2010.