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 that is built by extending the Jikes RVM. ByteSTM implements two STM algorithms, TL2 and RingSTM. We modify Jikes RVM's Optimizing compiler to transparently support implicit transactions. Being implemented at the VM-level, it accesses memory directly and handles memory uniformly, and avoids Java garbage collection by manually managing memory for transactional metadata. Our experimental studies reveal throughput improvement over other non-VM STMs in the range of 7%–66% on micro-benchmarks and 7%–76% 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 [19].

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 good performance [34] and is composable.

TM has been proposed in hardware (HTM [4, 11, 12, 16, 18, 37]), in software (STM [21, 22, 32, 35, 38]), and in combination (HybridTM [13, 23, 24, 26, 30, 41]). 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

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

Thread B

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

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 [20, 28, 39**?**] and those that use *implicit* transactions [8, 22, 31]. 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) to mark sections of the code as atomic. Instrumentation is used to add transactional code transparently to the atomic sections. Some implicit transactions work only with transactional objects [8, 31], while others work on any object and support external libraries [22].

Compiler-based STMs (e.g., Intel C++ STM compiler, AtomJava [21]) 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 runtime 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, 10, 17, 40]. In [17], 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 [15]. This means that inconsistent reads may occur before a transaction

Feature	Deuce [22]	JVSTM [8]	ObjectFabric [28]	AtomJava [21]	DSTM2 [20]	Multiverse [39]	LSA-STM [31]	Harris and Fraser [17]	Atomos [1] [10]	Transactional monitors [40]	ByteSTM
Implicit transactions	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓
All data types	✓	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓
External libraries	✓	✗	✗	✓[2]	✗	✗	✗	✓	✗[3]	✓	✓
Unrestricted atomic blocks	✗	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓
Direct memory access	✓[4]	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓
Field-based granularity	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
No GC overhead	✓[5]	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓
Compiler support	✗	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓ & ✗[6]
Strong atomicity	✗	✗	✓	✓	✗	✗	✗	✗	✓	✗	✗
Closed/Open nesting	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗
Conditional variables	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗

[1] Atomos is a HybridTM but the 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] Using non-standard library.
[5] Uses object pooling, which partially solves the problem.
[6] ByteSTM can work with or without compiler support.

Table 1. Comparison of Java STM implementations.

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. [10] 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 [22] 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 managing the memory needed for transactional support. Moreover, if TM is supported using HTM (as in [10]), 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 [7].

1.2 Contributions

Motivated by these observations, we design and implement a VM-level STM: ByteSTM (Section 2). ByteSTM implements two algorithms: TL2 [14] and RingSTM [36]. It writes directly to memory without using reflection (unlike [20]) or a non-standard library (unlike [22]). 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]

In ByteSTM, a transaction can surround any block of code, and is not restricted to methods only. Memory bytecode instructions (i.e., load/store operations) that are reachable from a transaction are translated so that the resulting native code executes transactionally. ByteSTM works with all data types, not just transactional objects, and thereby supports external libraries. Currently, ByteSTM does not support closed/open nesting, strong atomicity, or conditional variables; these are 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.

We conducted experimental studies, comparing ByteSTM with other Java STMs including Deuce [22], Object Fabric [28], Multiverse [39], DSTM2 [20], and JVSTM [8] (Section 3). Our results reveal that, ByteSTM improves transactional throughput over non-VM STMs by 13% to 70% on micro-benchmarks, and by 10% to

[1] False conflicts may still occur due to other implementation choices, e.g., TL2 lock table [14], read/write signatures [36].

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 Jikes RVM (version 3.1.2) [3] using the Optimizing compiler. 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, loop optimization). Jikes RVM has no interpreter, and bytecode must be compiled before execution. Building the Jikes RVM with production configuration gives performance comparable to the HotSpot server JIT compiler [29].

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. 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[2].

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 mode 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 [8, 22, 31] 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 [22].

In ByteSTM, 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 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-set, the write-set, and other STM algorithm-specific metadata. These 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 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.

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.

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 a base object plus an offset for all cases.

Using absolute addresses limits us to only non-moving GC (i.e., a GC which releases unreachable objects without moving reachable objects, like mark-and-sweep GC). 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

[2] Note that, this is a very simplified and abstract overview. Actual STM algorithms have more details.

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, 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 $O(n)$. 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. Then we 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. 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 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.

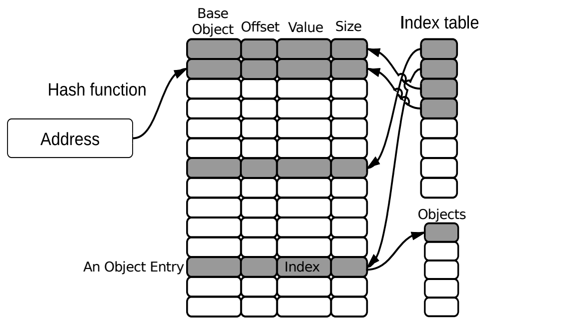


Figure 2. ByteSTM's write-set using open address hashing.

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 2 shows how ByteSTM's write-set is represented using open-addressing.

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 local variables are restored and the transaction can restart as if nothing has changed in the local variables. This technique simplifies the handling of local variables since there is no need to monitor them.

ByteSTM supports opacity [15]. When an inconsistent read is detected in a transaction, the transaction is immediately aborted. Then local variables are 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. Moreover, it can be optimized to a simple `goto` if the exception handler exists in the same method that throws the exception. This 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 [25]. 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 [22]. For example, object pooling is used to reduce the pressure on the memory system and improve performance in [22], 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. Within Jikes RVM, objects can be allocated manually in the immortal space, which is not monitored by the GC.

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 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., objects array). The object's index in the objects array is stored in the write-set entry value (Figure 2). Specifically, if an object contains another object (e.g., a field that is a reference), then 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 these base objects. This prevents the GC from reclaiming the objects.

2.6 STM Algorithms

ByteSTM's modular architecture allows STM algorithms to be easily "plugged in." We implemented the TL2 [14] and RingSTM [36] STM algorithms. 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.

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. (None of the Java STMs in Table 1 support irrevocable transactions.)

ByteSTM does not support closed/open 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) [27]. 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) [2, 5], a global data structure, where a child can find its parent's metadata, can be added.

3. Experimental Results

3.1 Test Environment

We conducted our experiments on a 48-core machine, which has four AMD Opteron$^{\text{TM}}$Processors (6164 HE), each with 12 cores running at 1700 MHz, and 16 GB of memory. The machine runs Ubuntu Linux 10.04 LTS 64-bit. Jikes RVM version 3.1.2 is used to run all experiments. We used the production configuration, which includes the Jikes Optimizing compiler and GenImmix GC [6] (i.e., a two-generation copying GC), which match ByteSTM configurations.

The competitor STMs include Deuce [22], ObjectFabric [28], Multiverse [39], and JVSTM [8]. Note that Deuce uses a non-standard proprietary API (i.e., `sun.misc.Unsafe`), which is not fully supported by Jikes RVM. To run Deuce atop Jikes RVM, we therefore added necessary methods to Jikes RVM's `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. 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 [9] benchmark suite[3] (Vacation, KMeans, Genome, Labyrinth, and Intruder). For the micro-benchmarks, 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 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.

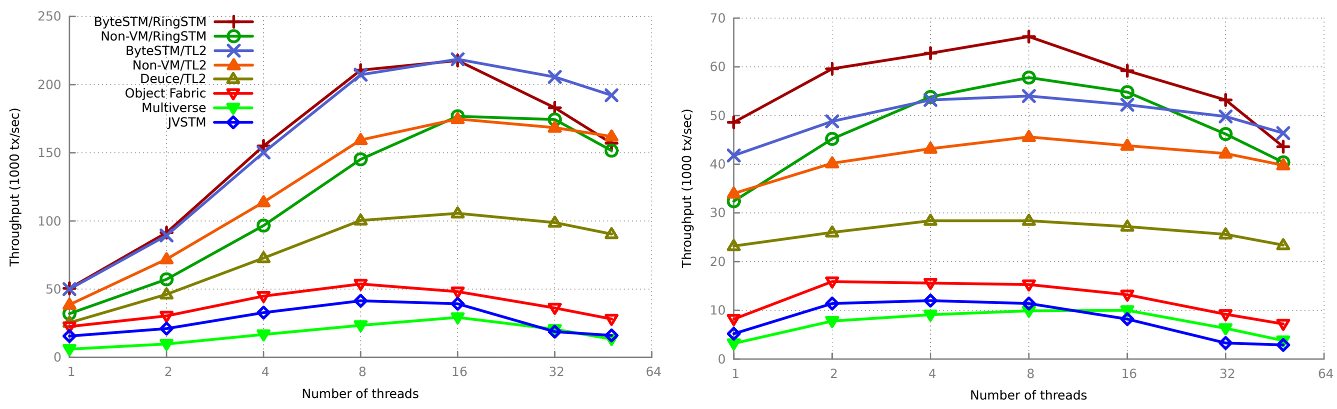
For each benchmark, we conducted experiments with different read/write ratios: 20% writes, 50% writes, 80% writes, and 100% writes. For brevity, we only show results for 20% and 80% writes; results were consistent for the other cases.

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 3 shows the throughput at increasing number of threads. ByteSTM has two curves representing the RingSTM and TL2 algorithms. ByteSTM/RingSTM achieves the best performance but did not scale well because of using a centralized data structure. ByteSTM/TL2 performance degrades as the ratio of writes in-

[3] We used Arie Zilberstein's Java implementation of STAMP, which is part of the Deuce project's open source repository.



(a) 20% writes. (b) 80% writes.

Figure 3. Throughput under Linked List.

creases. Deuce's performance is the best among other STMs. Other STMs perform similarly, and all of them have very low throughput.

At high contention, TL2's scalability degrades. ByteSTM/TL2 outperforms non-VM/TL2 by as much as 17% and up to 30%. ByteSTM/RingSTM outperforms non-VM/ RingSTM in the 15%–60% range. The gap between TL2 and RingSTM in high contention is due to the elimination of the read-set and usage of signatures in RingSTM, given the very small number of writes.

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 4 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. Other STMs' performance and scalability are poor.

ByteSTM/TL2 outperforms non-VM/TL2 in the 13–44% range. ByteSTM/RingSTM outperforms non-VM/RingSTM in the 10–33% 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.

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 5 shows the results. We observe that, in all cases, ByteSTM/TL2 achieves the best performance. RingSTM's performance begins to degrade with large number of 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. Deuce/TL2 is the third in performance, but also suffers from performance degradation after 16 threads.

ByteSTM/TL2 outperforms non-VM/TL2 in the 7–12% range. ByteSTM/RingSTM outperforms non-VM/RingSTM in the 7–14% range.

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 6 shows the results. ByteSTM/RingSTM achieves the best performance, 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 8–37% range. ByteSTM/RingSTM outperforms non-VM/RingSTM in the 7–26% range.

3.3 Macro Benchmarks

3.3.1 Vacation

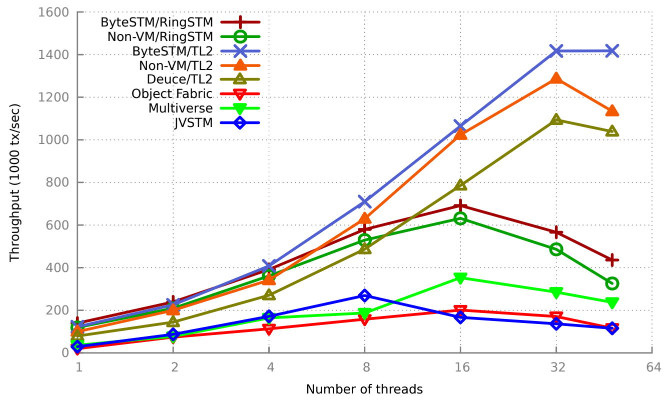
The Vacation benchmark [9] is characterized by medium-length transactions, medium read-sets, medium write-sets, and long transaction times (compared with other STAMP benchmarks). We conducted two experiments: one with low contention, and the other with high contention.

Figure 7 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, but it performs better at small number of threads. ByteSTM/TL2 outperforms non-VM/TL2 by an average of 8% in low contention and 22% in high contention. ByteSTM/RingSTM outperforms non-VM/RingSTM by an average of 36% in low contention and 108% in high contention.

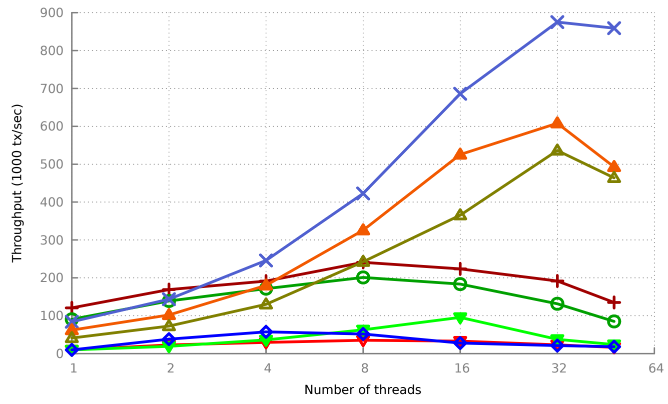
3.3.2 KMeans

The KMeans benchmark [9] 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 8 shows the results. We observe that ByteSTM/TL2 scales well in both cases and performs similar. ByteSTM/RingSTM performs well in low contention and small number of threads. ByteSTM/TL2 outperforms non-VM/TL2 by an average of 53% in

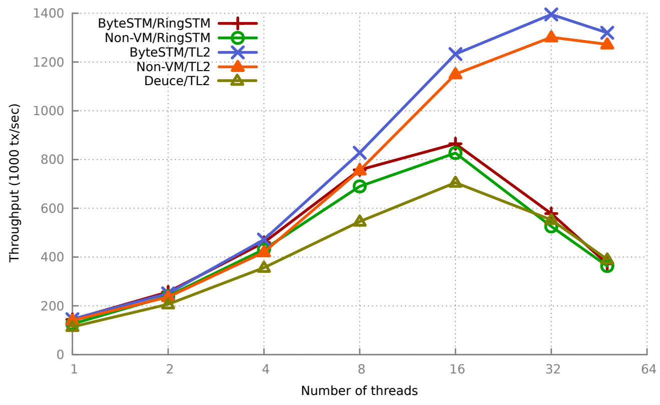


(a) 20% writes.

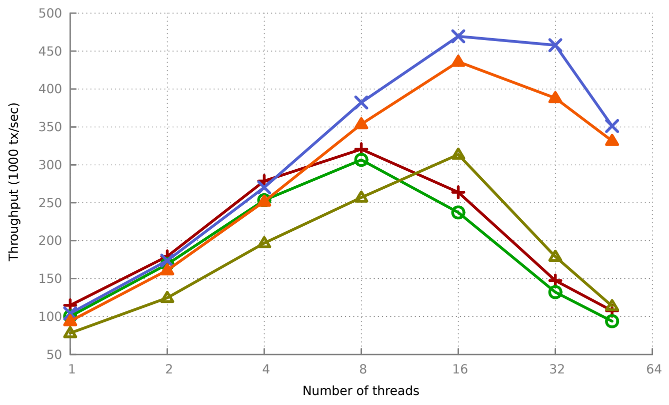


(b) 80% writes.

Figure 4. Throughput under Skip List.

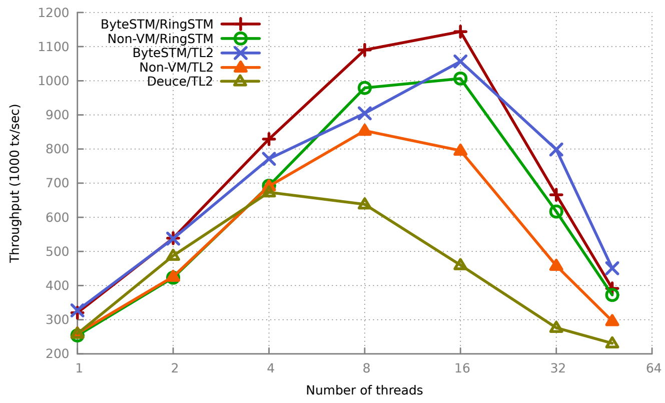


(a) 20% writes.

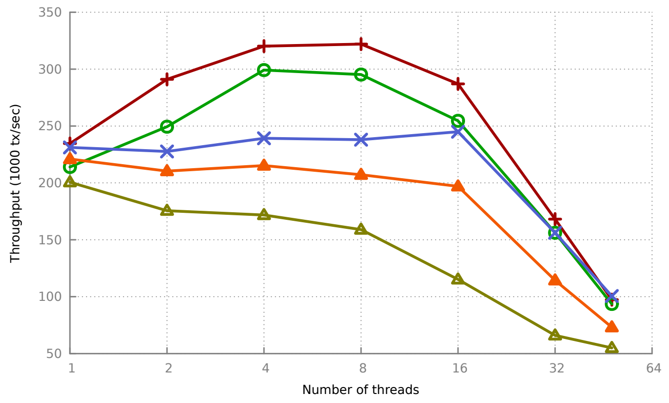


(b) 80% writes.

Figure 5. Throughput under Red-Black Tree.



(a) 20% writes.



(b) 80% writes.

Figure 6. Throughput under Hash Set.

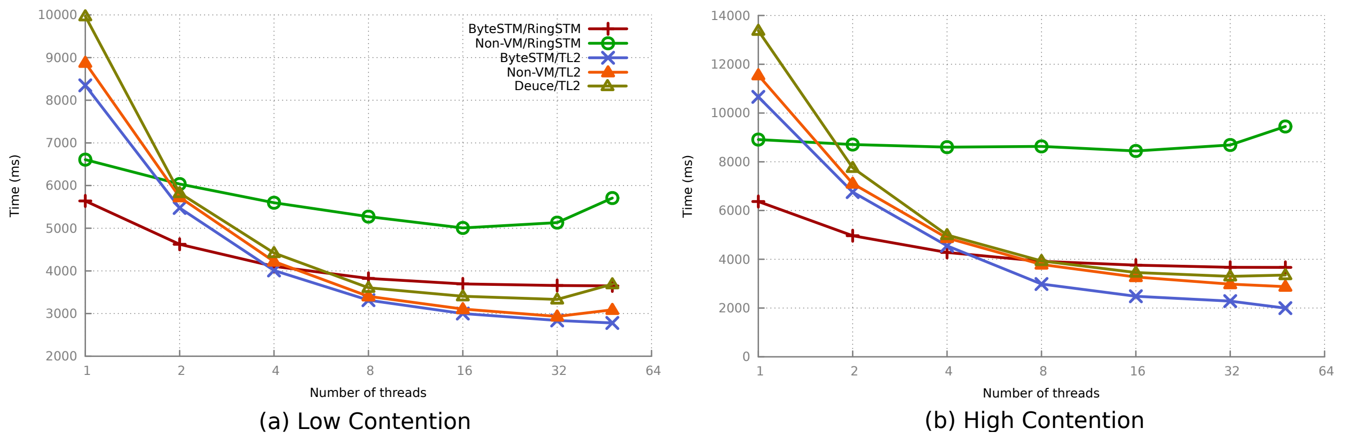


Figure 7. Execution time under Vacation.

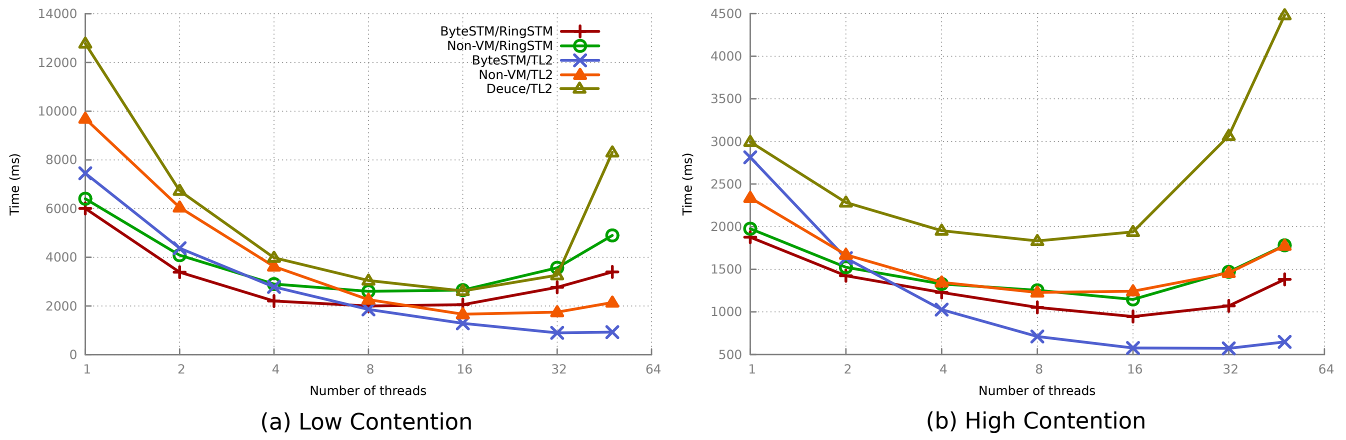


Figure 8. Execution time under KMeans.

low contention and 76% in high contention. ByteSTM/RingSTM outperforms non-VM/RingSTM by an average of 27% in low contention and 18% in high contention.

3.3.3 Labyrinth

The Labyrinth benchmark [9] is characterized by long transaction lengths, large read-sets, large write-sets, long transaction times, and very high contention.

Figure 9 shows the results. ByteSTM/TL2 achieves the best performance and scalability after 16 threads. ByteSTM/ RingSTM suffers from extremely high number of aborts due to false positives (large number of reads and writes) and long transactions, and shows no scalability. ByteSTM/TL2 outperforms non-VM/TL2 by an average of 11.5%. ByteSTM/RingSTM outperforms non-VM/RingSTM by an average of 8%.

3.3.4 Intruder

The Intruder benchmark [9] is characterized by short transaction lengths, medium read-sets, medium write-sets, medium transaction times, and high contention.

Figure 10 shows the results. We observe that ByteSTM/TL2 achieves the best performance. ByteSTM/RingSTM suffers from increased aborts due to false positives and does not scaler after 8 thread. ByteSTM/TL2 outperforms non-VM/TL2 by an average of 66%. ByteSTM/ RingSTM outperforms non-VM/RingSTM by an average of 7%.

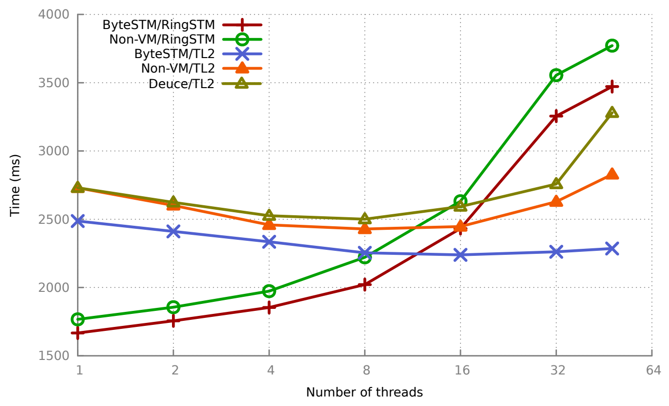


Figure 9. Execution time under Labyrinth.

Results for the Genome benchmark [9] are skipped for brevity; they are similar to the Vacation benchmark.

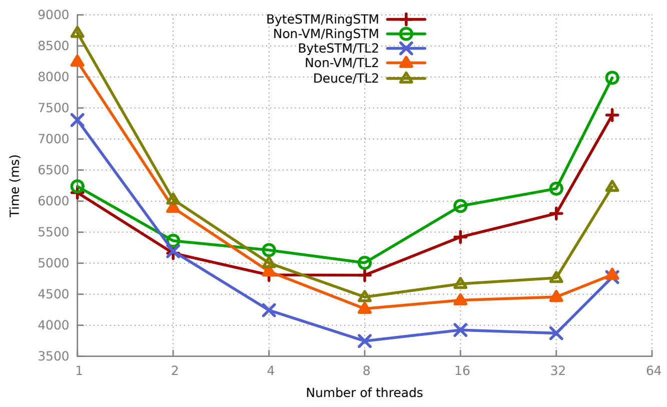


Figure 10. Execution time under Intruder.

3.4 Summary

ByteSTM improves performance over non-VM implementations by an overall average of 30%. On micro-benchmarks, ByteSTM improves by 7% to 66%. On macro-benchmarks, ByteSTM's improvement ranges from 7% to 76%. Moreover, scalability is significantly better. ByteSTM, in general, is better when the abort ratio and contention are high.

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. Conclusions

Our work shows that implementing an STM at the VM-level 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, and no instrumentation is required. 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 irrevocable operations (not currently supported) can be efficiently supported.

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

ByteSTM is open-sourced and is freely available at `hydravm.org/bytestm`. A modified version of ByteSTM is currently used in the HydraVM project [33], which is exploring automated concurrency refactoring in legacy code using TM.

Acknowledgements

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385.

References

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

[2] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *PPoPP*, pages 163–174, 2008.

[3] 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.

[4] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, pages 316 – 327, 2005.

[5] J. a. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapalka. Leveraging parallel nesting in transactional memory. In *PPoPP*, pages 91–100, 2010.

[6] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI*, 2008.

[7] 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.

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

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

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

[11] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *ISCA*, pages 227–238. IEEE, 2006.

[12] 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.

[13] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, pages 336–346. ACM, 2006.

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

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

[16] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *ISCA*. IEEE, 2004.

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

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

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

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

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

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

[23] Y. Lev and J.-W. Maessen. Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In *PPoPP*, pages 197–206. ACM, 2008.

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

[25] F. Meawad et al. Collecting transactional garbage. In *TRANSACT*, 2011.

[26] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: log-based transactional memory. In *HPCA*, pages 254 – 265, 2006.

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

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

[29] M. Paleczny, C. Vick, and C. Click. The Java Hotspot^{TM}Server Compiler. In *Java^{TM}Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[30] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA*, pages 494 – 505, 2005.

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

[32] T. Riegel, P. Felber, and C. Fetzer. TinySTM. `http://tmware.org/tinystm`, 2010.

[33] M. M. Saad, M. Mohamedin, and B. Ravindran. HydraVM: extracting parallelism from legacy sequential code using STM. In *HotPar*. USENIX, 2012. `hydravm.org`.

[34] 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.

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

[36] M. F. Spear et al. RingSTM: scalable transactions with a single atomic instruction. In *SPAA*, pages 275–284, 2008.

[37] J. Stone, H. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(4):58 –71, nov 1993. ISSN 1063-6552. doi: 10.1109/88.260295.

[38] University of Rochester. Rochester Software Transactional Memory. `http://www.cs.rochester.edu/research/synchronization/rstm/index.shtml`, `http://code.google.com/p/rstm`, 2006.

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

[40] A. Welc et al. Transactional monitors for concurrent objects. In *ECOOP*, 2004.

[41] L. Yen, J. Bobba, M. Marty, K. Moore, H. Volos, M. Hill, M. Swift, and D. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *HPCA*, pages 261 –272, 2007.