

Making Object-Based STM Practical in Unmanaged Environments^{*}

Torvald Riegel

Dresden University of Technology
torvald.riegel@inf.tu-dresden.de

Diogo Becker de Brum

Dresden University of Technology
diogo.becker@inf.tu-dresden.de

Abstract

Current transactifying compilers for unmanaged environments (e.g., systems software written in C/C++) target only word-based software transactional memories (STMs) because the compiler cannot easily infer whether it is safe to transform a transactional access to a certain memory location in an object-based way. To use object-based STMs in these environments, programmers must use explicit calls to the STM or use a restricted language dialect, both of which are not practical.

In this paper, we show how an existing pointer analysis can be used to let a transactifying compiler for C/C++ use object-based accesses whenever this is possible and safe, while falling back to word-based accesses otherwise. Programmers do not need to provide any annotations and do not have to use a restricted language. Our evaluation also shows that an object-based STM can be significantly faster than a word-based STM with an otherwise identical design and implementation, even if the parameters of the latter have been tuned.

1. Introduction

Software Transactional Memory (STM) is supposed to make it easier for programmers to harness parallelism in applications. One important part of this goal is that to use transactions, programmers should just have to declare where transactions start and commit rather than having to explicitly delegate every memory access to a STM runtime system. Instead, a compiler should transactify the application (i.e., transform raw memory accesses to calls to STM functions, for example).

Recent transactifying compilers in managed environments [1, 14] assume an object-based STM (i.e., for concurrency control, the granularity of an access is an object), whereas transactifying compilers for unmanaged environments [10, 3, 15, 11] target word-based STMs.¹ However, there is no evidence that word-based STMs are better than object-based STMs in unmanaged environments or vice versa. Instead, the choice taken by compilers for managed environments suggests that object-based accesses could have advantages over word-based accesses, and we would like to enable systems software that often does not run in managed environments to also benefit from potential object-based STM advantages. Section 2 contains a description of these advantages.

^{*}This is an updated version of the original paper with (1) an enhanced description of the Red-Black Tree benchmark, (2) corrected labels for the respective results, and (3) an updated version of Figure 2.

¹Informally, in unmanaged environments application code directly accesses the resources provided to its process. In managed environments applications access resources through an intermediate layer such as a Java Virtual Machine.

One reason why transactifying compilers for unmanaged environments typically did require word-based STMs up to now is that the information about which object a certain memory location belongs to is not easily available in unmanaged environments. In languages such as C and C++, type information is available in the source code but accesses are not guaranteed to be type-safe at runtime unless restricted language dialects are used.

In this paper, we show how a previously word-based transactifying compiler for C/C++ can support object-based STMs by using a compiler analysis that infers information about data structures in a program. The compiler does not require programmers to add any additional annotations to applications nor to use restricted language dialects.

Transactional accesses are transformed to be object-based with in-place or external STM metadata where possible; In cases where the compiler does not know whether this would be safe, word-based accesses are used. This compile-time decision is not fixed per data type but can differ for different instances of a type. All accesses to a certain instance are guaranteed to be either all word-based or all object-based.

We give more background information and describe related work in Section 2. The analysis and transformations executed by the transactifying compiler are discussed in Section 3. We shortly describe our object-based STM in Section 4, evaluate the applicability of the compiler and the performance of word-based vs. object-based STM accesses in Section 5 and conclude in Section 6.

2. Background and Related Work

The granularity of a transactional access determines the size of the entities that the transactional memory is composed of. It is an important design choice for STMs. Currently, there are two basic options: word-based and object-based accesses. The former divides memory into blocks of a certain size (e.g., of word or cacheline size), the latter assumes that memory consists of separate objects of different sizes.

STMs maintain metadata needed for concurrency control (e.g., locks) for these entities. In word-based STMs, memory locations are typically mapped to metadata using a hash function. In object-based STMs, metadata is either *external* and associated with the base address of the object (i.e., the smallest memory address that is within the object) using a hash function, or it is kept *in-place* by embedding it into every transactionally accessed object. Determining the base address is difficult in unmanaged environments because there, accesses might target any memory location and not only fields of a known object as is typically the case in managed environments. Likewise, word-based STM interfaces for accesses just require the memory location as parameter, whereas object-based STMs additionally require the base address of the object that is targeted by an access.

Most of the STMs for unmanaged environments that we are aware of are word-based [9, 3, 7]. Those that are object-based [8, 12] only provide their service as a library and cannot be transparently used by programmers (i.e., transactional memory accesses have to be explicitly requested), which unfortunately limits the impact that they can have in practice. Also, a lot of recently proposed STMs use a combination of invisible reads and locking for writes to keep runtime overheads small.

2.1 Potential Advantages and Disadvantages of Object-Based STMs

If object-based STMs use in-place metadata, they can potentially benefit from the improved locality compared to when using external metadata. If objects are not too large, the object data and metadata will likely reside in the same cacheline, which reduces the cache footprint of transactions. It can also result in fewer cache misses when reading data modified by other transactions because there is only one cache miss for both object data and metadata and not two. Also, no indirection is necessary to access the metadata. Even with external metadata, the implicit partitioning of objects can be useful, for example to prevent having to get several locks to write to a single object.

However, the very same properties can also decrease the performance, depending on the workload. For example, if there is very little contention, using only a few locks could yield the highest STM performance but this kind of tuning is not possible with the fixed object to metadata mapping in the case of in-place metadata. Similarly, in-place metadata increases the size of objects and can thus increase cache footprint. Another example is that to release memory, it is for typical word-based STM designs sufficient to just update all metadata that the object maps to; in case of an object-based design with in-place metadata, the metadata must remain type-stable until other transactions cannot reach it anymore, so the STM cannot release the memory immediately but must handle these cases specially.

2.2 Related Work: Object-based STMs

Two object-based STMs [14, 1] have been recently proposed for managed environments and show that the overhead of these STMs can be quite small. Both use invisible reads and locking for writes but do not guarantee the consistency of a transaction's readset at runtime, which we believe is essential in unmanaged environments.

Recent proposals for object-based STMs suitable for unmanaged environments include NZTM and RSTM. NZTM [8] tries to exploit the performance advantages of blocking accesses while switching to a nonblocking in case of contention. It uses four pointers of STM metadata per object but does not require any indirection if there is no contention. RSTM [12] is a nonblocking object-based STM that contains implementations for different validation and conflict detection strategies.

McRT-STM [2] supports object-based conflict detection for small objects by using a custom memory allocator that allocates small memory chunks in a special memory region. Chunks are grouped in memory blocks together with other chunks of the same size. The STM has to check at runtime whether a target address of a transactional access is in the memory region reserved for small objects. If it is, then it computes the base address of the object by loading the size of objects in this memory block from the block's header. Compared to our approach, McRT-STM can support object-based accesses without having to rely on a pointer analysis of the source code. However, it has a higher runtime overhead because of the additional load and address check, which have to be performed for each transactional access in the worst case.

2.3 Related Work: Transactifying Compilers

A couple of transactifying compilers for unmanaged environments have been introduced lately [10, 3, 15, 11]. They all can transform C/C++ programs, target word-based STMs, but expect different styles of transaction declarations.

Tanger [10] uses the LLVM compiler infrastructure [6] to transform applications at the level of LLVM's intermediate representation, expects transaction begin and commit declarations in the form of calls to special marker functions, and is available under an open-source license. It reuses LLVM's general purpose optimization passes but does not perform STM-specific optimizations. Wang et. al. present several STM optimizations for unmanaged environments in [3] and show that these reduce runtime overheads significantly. OpenTM [15] is based on the GNU Compiler Collection and transforms transactions that are declared with custom OpenMP constructs. Damron et. al. briefly describe another compiler in [11].

Recent compilers for managed environments [1, 14] use STM-specific optimizations and target object-based STMs that are tightly integrated with the already "object-based" managed runtime environment. [1] can select word-based or object-based accesses per data type. Optimizations for strong isolation are described in [13].

3. Automatic Object-Based Transactification

To be able to automatically transform memory accesses to calls to an object-based STM runtime, we need to know whether a certain transactional access in the program targets an object. In our context, an object is a continuous chunk of memory that can be defined and identified by its size (most likely derived from its type) and its base address in memory (i.e., the start of the memory chunk). We also want to only consider objects that could correspond to user-defined data structures in the program (e.g., ignore primitive types because they seem too small to justify object-based accesses).

To make such transformations safe, the compiler must ensure that during the lifetime of any memory chunk (either dynamically allocated, on the stack, or global), all accesses to memory locations in the chunk agree on whether such locations are part of an object, and if so, which one.

However, this is not trivial in unmanaged environments. We focus in what follows on C/C++ programs. In these, for example, programmers can access every memory location, casts between types are common, and pointers to fields in objects can be taken and passed to other functions.

What we therefore need is an inter-procedural analysis that computes points-to graphs for the complete program. We use the *Data Structure Analysis* (DSA) [5], a context-sensitive, unification-based, and field sensitive pointer analysis that can identify instances of data structures and important properties of those (e.g., type safety). Context sensitivity means that data structures get distinguished based on call graphs and not just allocation sites, for example. Unification refers to pointers targeting at most one node in the points-to graphs, which makes the analysis fast and scalable. Field sensitivity is distinguishing between the different fields in data structures. DSA can handle calls to external functions, so we do not need to analyze the complete program, although more information of course yields better analysis results. DSA is implemented as an analysis pass in the LLVM compiler framework.

In a nutshell, automatic object-based transactification takes the following steps: we (1) compile the program using LLVM and link all parts of it into a single module, (2) use DSA to analyze this module and infer the information needed to decide between word-based and object-based accesses, (3) enlarge allocations for data structures that are potential targets of object-based accesses if the STM uses in-place metadata, and (4) transform the transactional

parts of the application, choosing between word-based and object-based accesses according to the DSA results for each data structure instance.

We have implemented this on top of Tanger [10], which also uses LLVM. We use Tanger’s word-based transformations as fall-back for all accesses that cannot be safely transformed to be object-based. The transformations target a typical STM interface that, however, has functions for both word-based and object-based accesses. The STM implementation is discussed in Section 4. Next, we describe the analysis and the transformations in more detail.

3.1 Analysis

DSA models points-to information as a graph of nodes that represent data structures (DS). It builds this DS graph incrementally by first analyzing each function and determining properties of nodes based on how pointers and data structures instances are used. For example, if the program contains a load to an address calculated as a pointer plus a field offset derived from a certain structure type, then the pointer is assumed to point to a DS node with this type. Besides type and points-to information for pointers, DS nodes also contain a set of flags stating whether the data structure is on the stack or on the heap (derived from where the pointer originated), whether the information about the node is complete (i.e., all of its uses have been analyzed), and a few other things.

After this function-local phase, DS graphs contain complete information about DS nodes that do not (transitively) escape from the function. DSA then executes a bottom-up pass on the program’s call graph and merges the DS graphs of callees into callers, which results in complete information for nodes that do not escape to callers. DS graphs are merged by merging the information in aliasing DS nodes into a single node. Finally, DSA makes a top-down pass on the call graph to propagate information from callers to callees. Note that nodes that escape to non-analyzable functions (e.g., external functions) or through external globals will remain marked as incomplete. Also, nodes are merged in a way that preserves uncertainty. For example, if a pointer with a DS node with type A escapes to a calling function which uses the pointer as incompatible type B (i.e., the pointer would have to point to two different nodes), then DSA will infer that the type for the pointer is not known in *any* of the functions; the node’s information is “collapsed” and A and B will point to a “collapsed” node. There exists cases in which the current DSA implementation cannot ensure consensus on node information with just the bottom-up and top-down passes and should instead merge more often. However, we have not observed this in the benchmarks we tried (see Section 5), and we see this as an implementation issue and not a conceptual limitation.

Figure 1 shows a part of the DS graph for the `main` function of STAMP’s [4] Vacation benchmark.² It shows some of the important data structures used in this benchmark. The small boxes in the bottom of each node represent one field in the respective structure types. Edges between the nodes represent points-to information (e.g., the first and fourth field of structure `manager_t` point to two separate red-black tree instances). The analysis of all shown nodes is complete. All nodes except the one marked as “collapsed” have a type, meaning that all memory locations associated with such a node are accessed in a type-safe way in the program. Only the node shown at the top is an array, meaning that all other nodes originate from non-array allocations and uses.

We classify nodes as safe or not safe for object-based accesses based on the DS graphs computed by top-down DSA. A node can be accessed using object-based STM functions iff it is of known

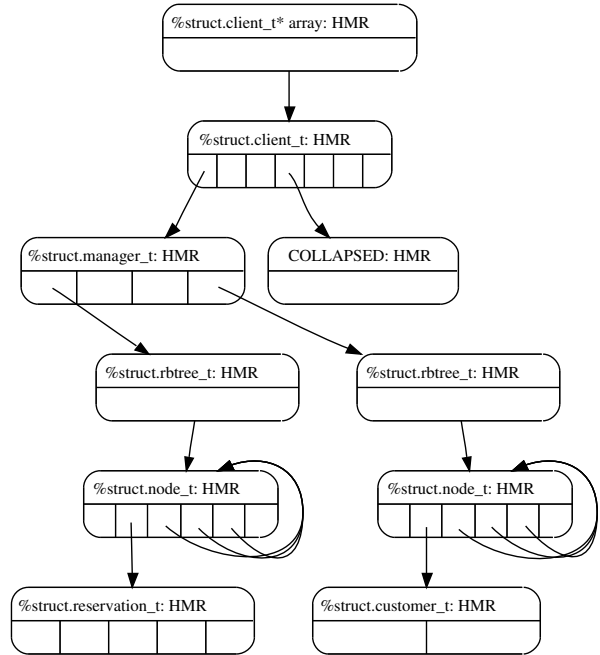


Figure 1. A part of the DS graph for function `main` of STAMP’s Vacation benchmark.

structure type and accessed in a type-safe way, has been completely analyzed, is not external, and is not an array. We will discuss these constraints again after describing the object-based transformations.

3.2 Transformation

Our compiler transformations rewrite transactional memory accesses into calls to STM functions with typical load/store interfaces (i.e., function arguments are transaction descriptors, addresses, and values). However, the object-based variants of these functions additionally require the base address and the size of the object as arguments.

When transforming a memory access, we determine the size of the object based on the type of the DS node associated with the target of the access. The base address is computed by either reusing base pointers available in the program³ or based on the offset information contained in the points-to edges in DS graphs (pointers can target specific fields of objects).

To be able to support STMs that use in-place metadata (i.e., embed transactional metadata such as locks in each object), we have to make sure that there is enough space for the metadata. We currently append metadata to the end of the object (i.e., at base address plus object size) but prefixing the object with metadata would also be possible. To reserve sufficient space, we enlarge all allocations (on stack and on heap) that could be used in an object-based access by the size of the metadata and let the STM initialize the metadata after the allocation. We also redirect calls that release object-based memory in transactions to object-based variants of the respective STM functions (we explain the reason for that in Section 4).

Safety The basis for the safety of these transformations are the guarantees given by DSA and the properties that are required for

²The complete graph is significantly larger, and also contains more edges starting at the nodes shown in the figure.

³Address computations for pointers to structure fields are explicit instructions in LLVM’s intermediate representation for code. They start at a pointer and navigate through the data structure definitions contained in the program.

DS nodes to be eligible for object-based accesses. We only consider typed data structures with stable DS node information that all parts of the program agree on. We make sure that there is no unknown behavior, constraints or uses for each object-based node by requiring complete node information. This also makes sure that memory for objects was allocated by a function that our compiler transformations know how to handle.⁴ Data structure instances that are part of an array are not considered as potentially object-based accesses, so allocations reserve memory for only one object and metadata does not need to be explicitly added to data type definitions. We only treat data structures with structure types as object-based to filter out primitive types.

4. STM Implementations

In this section, we briefly describe our object-based STM implementation. It is based on *TinySTM* [9], a time-based, word-based, blocking STM. An array with L versioned locks is used for concurrency control, and memory addresses are mapped to locks based on a hash function that first shifts off a certain number (S) of less significant bits of the memory address and then uses the remaining bits as offset into the array (modulo L).

The analysis used by our compiler ensures that object-based and word-based memory accesses are separated at compile time, which makes it easy to support both in an STM. We extended *TinySTM* to support object-based accesses by adding functions for object-based loads/stores to memory and for in-transaction release of memory. To allow for a better comparison to the original *TinySTM*, we chose to implement concurrency control for object-based accesses with the same algorithm as used in *TinySTM* for word-based accesses (time-based and blocking). Nevertheless, our approach should be applicable to other word-based and object-based STMs because of the compile-time separation between object-based and word-based accesses. How much a particular word-based STM implementations would benefit from object-based accesses depends on many different factors and is outside the scope of this paper. However, *TinySTM* is similar to other current STMs such as TL2 [7], so our results give an indication how much these STMs could benefit.

We implemented two variants of support for object-based accesses. *TinySTM-Obj* uses in-place metadata appended to objects. Metadata consists of a single lock word for each object (same layout as locks in the lock array, see [9] for details). The address for the lock is computed based on the base address and the size of the object that is supplied by the compiler. *TinySTM-ObjE* uses external metadata, namely the same array of locks that are used for word-based accesses and the same hash functions. However, it always selects the lock that the base address of the object maps to. Note that using the same set of locks for word-based and object-based accesses is possible in our implementation because object-based accesses reuse the STM's word-based logging and bookkeeping functions, which eases the implementation significantly. Nevertheless, this is not necessary, and we could have also integrated other existing object-based STMs.

When using in-place metadata, releasing object-based memory needs to be treated specially by the STM. Current word-based, time-based STMs update all locks (or version numbers) that are associated with any location in the the memory chunk that is to be released. This prevents concurrent readers and writers from accessing this memory chunk unless they revalidate their readset, which causes them to notice that, for example, the memory is not reachable anymore through pointers. After updating the locks and committing, the memory chunk can be immediately released.

⁴External functions would result in external and incomplete information, and manually forging memory chunks would either lead to nodes being part of arrays or no type information being available.

This is not possible with in-place metadata because metadata needs to be type-stable. The STM therefore keeps lists with to-be-freed memory chunks per thread and only releases memory after all other transactions that could still access the data have aborted or committed. This holds in a time-based STM as soon as the start timestamps of all other transactions are larger than the commit timestamp of the transaction releasing the memory.

All three implementations try to avoid redundant entries in the read set by logging an access only if the previous access targeted a different lock.

5. Evaluation

We evaluate two aspects of our approach in this section. First, we look at whether standard STM benchmarks contain object-based accesses and whether our transactifying compiler can detect and transform these accesses. Second, we evaluate the performance of the benchmarks when using the new object-based STM implementation in comparison to the original word-based *TinySTM* implementation.

We use the Vacation, KMeans, and Genome benchmarks from the STAMP benchmark distribution [4], version 0.9.5. We also use two classical microbenchmarks: integer sets implemented using (1) a red-black tree or (2) a sorted linked list, in which transactions repeatedly insert, remove, or look up elements (see [9] for details).

5.1 Object-Based Transactification

Table 1 shows compile-time (static) and run-time transformation results for the different benchmarks. Static loads and stores are instructions in the program that access memory from within a transaction and are replaced by the compiler with calls to the STM. The number of loads and stores at runtime were counted by instrumenting the STM and running the benchmark with a single thread in the default configuration for at least 10 seconds. Thus, the first columns give an indication of how many object-based accesses are in programs, and the last column shows whether these are important for the benchmark's performance due to being executed often.

Both the red-black tree and the linked list access a single logical⁵ linked data structure transactionally, and our compiler detects this and uses just object-based accesses.

STAMP's Vacation benchmark simulates a reservation system. Its transactions mostly operate on a couple of red-black trees and linked lists. With the original source code of the benchmark, DSA was not able to infer data structure information for the trees because the tree implementation uses keys and values of types integer but the benchmark uses these value integers to store pointers. We changed the implementation of the tree to be properly typed (void pointers are sufficient) because the original code is likely to be hardly portable and error-prone.⁶ We also expect that production-quality code does not contain such code. DSA could have inferred that the values are not integers but pointers by, for example, proving that the values are not modified by integer arithmetic but this is not yet implemented. After this change, DSA detects the tree and linked list data structures and only a few nodes with no information remain (see Figure 1). The majority of transactional accesses at runtime is object-based.

The KMeans benchmark mostly operates on arrays of primitive types. DSA detects these arrays but they are not considered for object-based transformations.

⁵DSA usually detects more data structures because it distinguishes, for example, between the head of the red-black tree and its nodes (see Figure 1).

⁶In fact, STAMP includes a bug fix for 64-bit architectures that increases the type of the integer from `int` to `long`.

Benchmark	Number of static WB loads/stores	Number of static OB loads/stores	Percentage of OB loads/stores (static)	Percentage of OB loads/stores at run-time
Red-Black Tree	0 / 0	84 / 77	100% / 100%	100% / 100%
Linked List	0 / 0	16 / 6	100% / 100%	100% / 100%
STAMP Vacation	25 / 9	243 / 188	90% / 95%	97% / 84%
STAMP KMeans	9 / 7	0 / 0	0% / 0%	0% / 0%
STAMP Genome	47 / 19	32 / 18	40% / 48%	12% / 98%

Table 1. Applicability of object-based compiler transformations (OB is object-based, WB is word-based, single-threaded benchmark runs, counting only transactional loads/stores).

The Genome benchmark’s transactions access a mix of linked data structures and a large number of character strings. Unlike the manual instrumentations in STAMP for STMs, our compiler also treats the frequent string comparisons in the benchmark as transactional accesses. Word-based accesses are used for the strings, which leads to the low percentage of object-based accesses at run-time compared to the percentage of static object-based accesses in the program shown in Table 1.

Overall, our compiler is able to detect and transform the majority of object-based accesses in these benchmarks with the exception of the integer-to-pointer casting tree in the original Vacation benchmark. Special support for arrays could allow for further STM optimizations in KMeans and Genome.

5.2 STM Performance

To evaluate the performance of object-based versus word-based accesses, we measured transaction throughput for TinySTM, TinySTM-Obj, and TinySTM-ObjE on a two-way Intel quad-core machine. We compiled all benchmarks as 32-bit executables using Tanager with the object-based transformations being enabled only for TinySTM-Obj (in-place metadata) and TinySTM-ObjE (external metadata). Every benchmark uses eight threads to execute transactions.

The performance of word-based accesses in TinySTM and object-based accesses that use external metadata is influenced by the parameters of the lock array and the hash function that maps memory locations to locks. Therefore, we show results for different numbers of locks and shifts (see Section 4).

We first present results for the two microbenchmarks because these do not use any word-based accesses. Figures 5 and 6 show results for the red-black tree, update rates⁷ of 20% and 80%, and trees with 512 and 64k elements on average. The integer elements are randomly chosen from a range between 0 and 64k. In the red-black tree, we initially add 256k elements but due to this range the tree will not contain more than 64k elements. Figure 7 shows the performance for the linked list with 20% or 80% updates and 512 or 16k elements. The figures also show isolated results for a few interesting configurations (e.g., a varying number of locks together with a fixed number of shifts).

The throughput results for TinySTM-Obj form a plane because this STM doesn’t access the lock array but only the locks embedded into the objects and is thus not affected by the number of locks or shifts.⁸ In the red-black tree benchmark, TinySTM-ObjE is in general better than or equal to TinySTM and seems to be less vulnerable to bad lock/shift settings. However, for small trees TinySTM-ObjE can only reach the performance of TinySTM-Obj

by using one particular good lock/shift setting, but is slower than TinySTM-Obj in all other cases. For large trees and a 20% update rate, TinySTM-Obj performs much better than the other STMs except when the number of locks is very high. With frequent updates and large trees, TinySTM-Obj performs worse than the other STMs unless the number of locks and shifts is completely disadvantageous; we cannot yet give a conclusive explanation for this behavior. Figure 6 compares performance when varying the number of shifts. An interesting observation is that a low number of shifts is required for small trees, whereas a higher number of shifts increases performance for large trees. For example, a large tree with 20% updates requires eight shifts to be as good as TinySTM-Obj; but if this setting is chosen, then small trees in the same application will perform considerably worse.

In the four configurations of the linked list that we measured, TinySTM-Obj has the highest throughput in almost all of the cases. TinySTM-ObjE can handle a small number of shifts better than TinySTM but overall, both require a specific lock/shifts setting to reach maximum throughput. Performance drops considerably as soon as parameters deviate from this setting. The peak for 2^{28} locks (which corresponds to 1 GB of STM metadata in 32-bit systems) is related to how the default memory allocator⁹ on our platform allocates memory. Every thread can allocate memory from a per-thread region to avoid contention; however, these regions are located very far away from the heap’s main region in the virtual address space. When both the number of locks and the distance of the beginnings of the regions are a power of two and the number of locks times the lock granularity is smaller than the distance, false conflicts can happen even if the number of locks seems to be large. This could be avoided by either changing the memory allocator or using a more advanced hash function. The former does not help if applications use custom allocators or if transactions access other data that is mapped to high memory addresses; the latter is difficult because the hash function is on the fast-path of each transactional access. Nevertheless, the number of shifts would have to be tuned to reach the highest performance.

The results highlight that STMs need to tune the settings for lock arrays to reach good performance. Furthermore, the red-black tree results show that workload properties such as the frequency of updates or the size of data structures can determine which lock/shift settings allow for the best performance. Object-based accesses with in-place metadata (TinySTM-Obj) are not affected by this and reach the best throughput on average, even if the other STMs are well-tuned. Better contention management could perhaps minimize the throughput differences caused by false contention, but would still not be sufficient to reach the same level of parallelism in every case.¹⁰

⁷ Inserting and removing elements are categorized as update transactions but might not update the data structures if the element to be added already exists or the element to be removed does not exist. Lookups are always read-only transactions.

⁸ Note that both benchmarks only contain object-based transactional accesses.

⁹ We used the standard allocator of the GNU C library.

¹⁰ Advanced contention managers could schedule transactions proactively to minimize the probability of conflicts, but they cannot avoid all possible (false) conflicts.

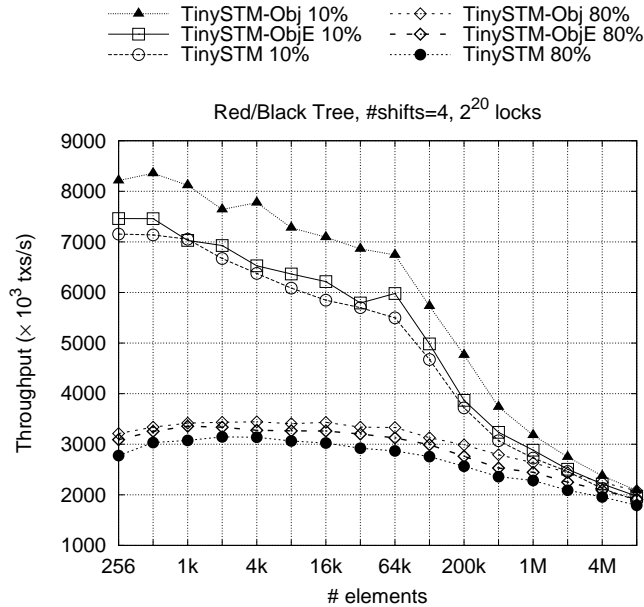


Figure 2. Red-black tree performance for different working set sizes.

Figure 2 shows the performance of the STMs for differently sized red-black trees with 10% and 80% update rates¹¹, respectively. For this benchmark, integer elements are randomly chosen from $[0, 2^{31})$. Note that the height of the balanced red-black trees increases very slowly compared to the number of nodes, and thus the number of transactional accesses also increases slowly while the working set size gets much larger. With 80% updates, throughput is limited by contention in the tree. Performance decreases for all STMs as soon as the tree does not fit into caches anymore. TinySTM-Obj outperforms the other STMs for 10% updates. However, we expected that the better locality due to in-place metadata would lead to higher performance advantages of TinySTM-Obj for larger working set size.

Figure 3 shows results for STAMP’s Vacation benchmark executed with default parameters for one million transactions. In contrast to the two microbenchmarks, transactions in Vacation consist of object-based and word-based accesses. This explains the substantially lower throughput of TinySTM-Obj with a few of the lock/shift settings. The results suggest that the few word-based accesses suffer heavily from false sharing in the lock array if the hash function does not cover the working set (i.e., the product of the number of locks and two to the number of shifts is too small). TinySTM-ObjE is slightly faster than TinySTM but both are significantly slower than TinySTM-Obj. TinySTM-Obj reaches its maximum throughput with a much smaller number of locks and is able to sustain this throughput, whereas the other STMs reach their performance maxima in only some of the lock/shift settings. Furthermore, a very high number of locks such as 2^{28} does not improve the performance of TinySTM and TinySTM-ObjE. This indicates that the better performance of TinySTM-Obj is not just caused by fewer false conflicts. For example, the different data structure instances in the benchmark could prefer different lock/shift settings when using TinySTM or TinySTM-ObjE (see Figure 6), which prevents optimal tuning for all data structures as there is only a single lock array.

¹¹ Inserting and removing elements are update transactions.

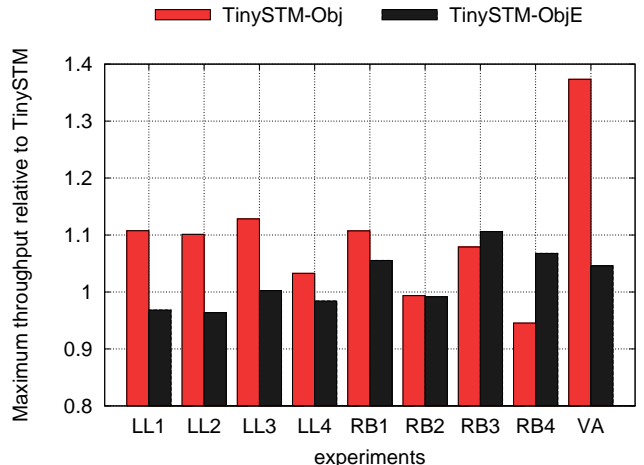


Figure 4. Performance of object-based accesses relative to word-based accesses (red-black tree configurations as in Figure 5, linked list as in Figure 7, and Vacation).

In contrast, object-based accesses do not depend on this particular tuning.

We also measured STM performance in the Genome benchmark. TinySTM-Obj often reaches the same transaction throughput independently of the numbers of shifts and locks, although Table 1 shows that the majority of loads are word-based. This indicates that these loads target the large amount of character strings used in the benchmark, and that conflicts usually only happen between the object-based accesses to the linked data structures. In contrast, the performance of TinySTM and TinySTM-ObjE (which both use a single lock array for the strings and the data structures) depends significantly on the number of locks and shifts. However, they are as fast as TinySTM-Obj for several favorable lock/shift settings. The variability of the performance is quite high in general and the overall performance of Genome with these settings seems to depend more on other factors such as contention management than on whether object-based accesses are used. Nevertheless, this highlights that a single lock array that is too small or too coarse-grained can easily lead to false conflicts, and that object-based accesses are not affected by this.

We did not perform any measurements for KMeans because it contains no object-based accesses.

Figure 4 summarizes the effect of object-based accesses on performance. It compares the maximum throughput that each of the STMs was able to reach when using the most favorable lock/shift setting¹² for the respective benchmark and STM. Thus, it considers improvements due to tuning of the lock array parameters but also shows limitations of tuning. One can see that object-based accesses with in-place metadata (TinySTM-Obj) are, with two exceptions, never worse than word-based accesses in our benchmarks and are often more than 10% better.

6. Conclusion

We have shown how a transactifying compiler can use pointer analysis to make object-based transactional accesses practical in unmanaged environments. We believe that this is important because STMs that just offer a library interface are only useful in a few scenarios and will not be useful to most programmers; STMs that

¹² The settings with 2^{28} locks were not considered because the space overhead would be too high for real applications.

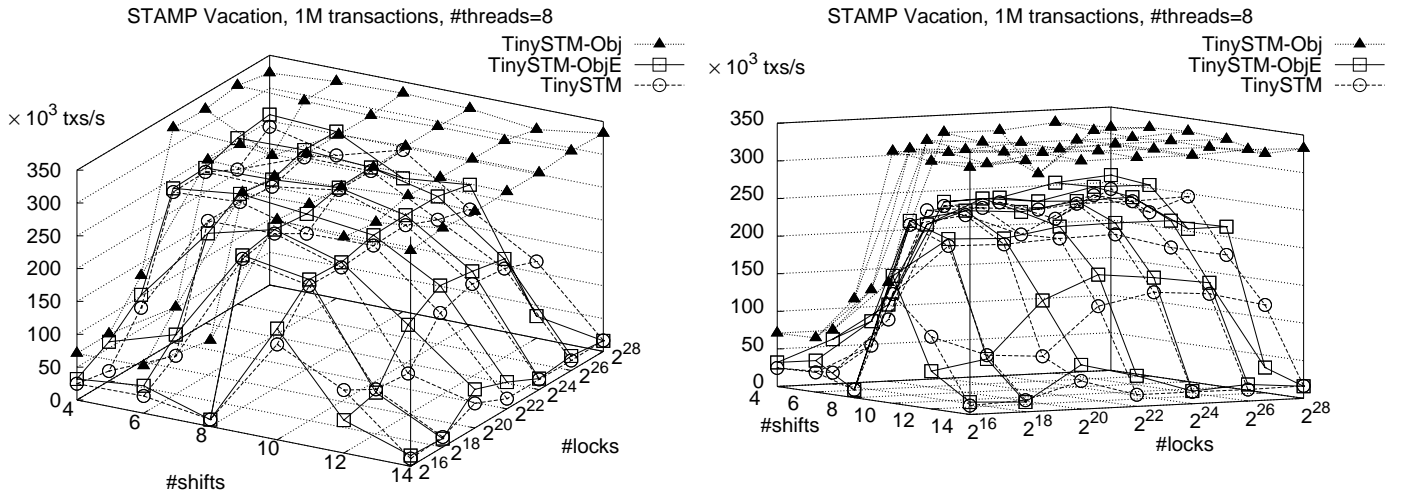


Figure 3. Transaction throughput for STAMP's Vacation benchmark (both figures show the same data).

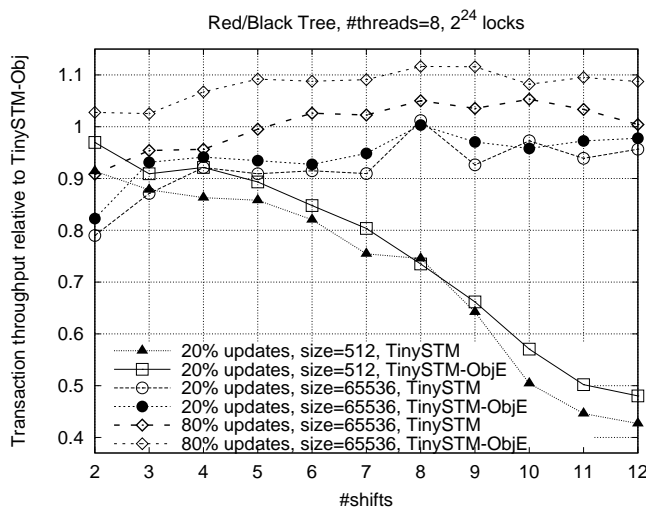


Figure 6. Transaction throughput for the Red-Black Tree benchmark.

compilers cannot or do not generate code for are not practical and might become extinct. Second, a lot of systems software and also a large percentage of the typical desktop applications are executed in unmanaged environments, which makes it important to support these environments. Third, object-based accesses have a number of potential performance advantages, and we showed in the evaluation that these advantages enable significant performance improvements in practice. Most importantly, STMs that can use object-based accesses require less tuning and can still reach a better performance than purely word-based STMs.

Although we only investigated STMs of one particular kind with a limited number of benchmarks, we think that our results indicate that STM designs should not limit themselves to only word-based accesses, and that compilers for unmanaged environments are indeed capable of finding and transforming object-based accesses in real programs.

We plan to release the object-based transformations in a future release of Tanger (<http://tinystm.org>).

Acknowledgements We thank Andrew Lenharth and Vikram Adve for answering our questions about DSA.

References

- [1] Ali-Reza Adl Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM Press.
- [2] Bratin Saha, Ali-Reza Adl Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.
- [3] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *International Symposium on Code Generation and Optimization (CGO)*, 2007.
- [4] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *34th Intl. Symposium on Computer Architecture (ISCA)*, 2007.
- [5] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. See <http://llvm.cs.uiuc.edu>.
- [6] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.
- [7] David Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In S. Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2006.

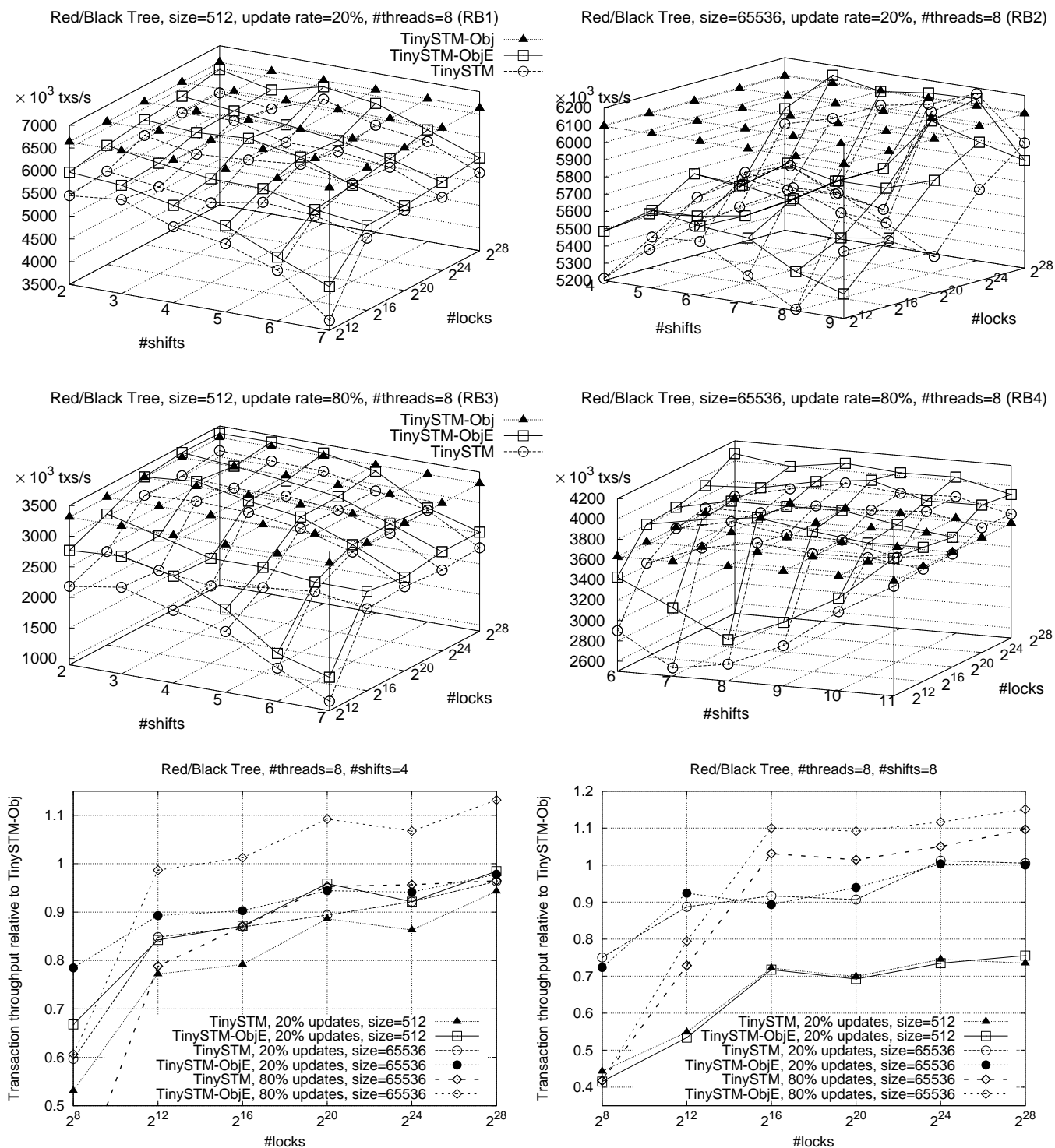


Figure 5. Transaction throughput for the Red-Black Tree benchmark.

[8] Fuad Tabba, Cong Wang, James R. Goodman, and Mark Moir. NZTM: Nonblocking Zero-Indirection Transactional Memory. In *TRANSACT*, 2007.

[9] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Perform-

mance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[10] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin

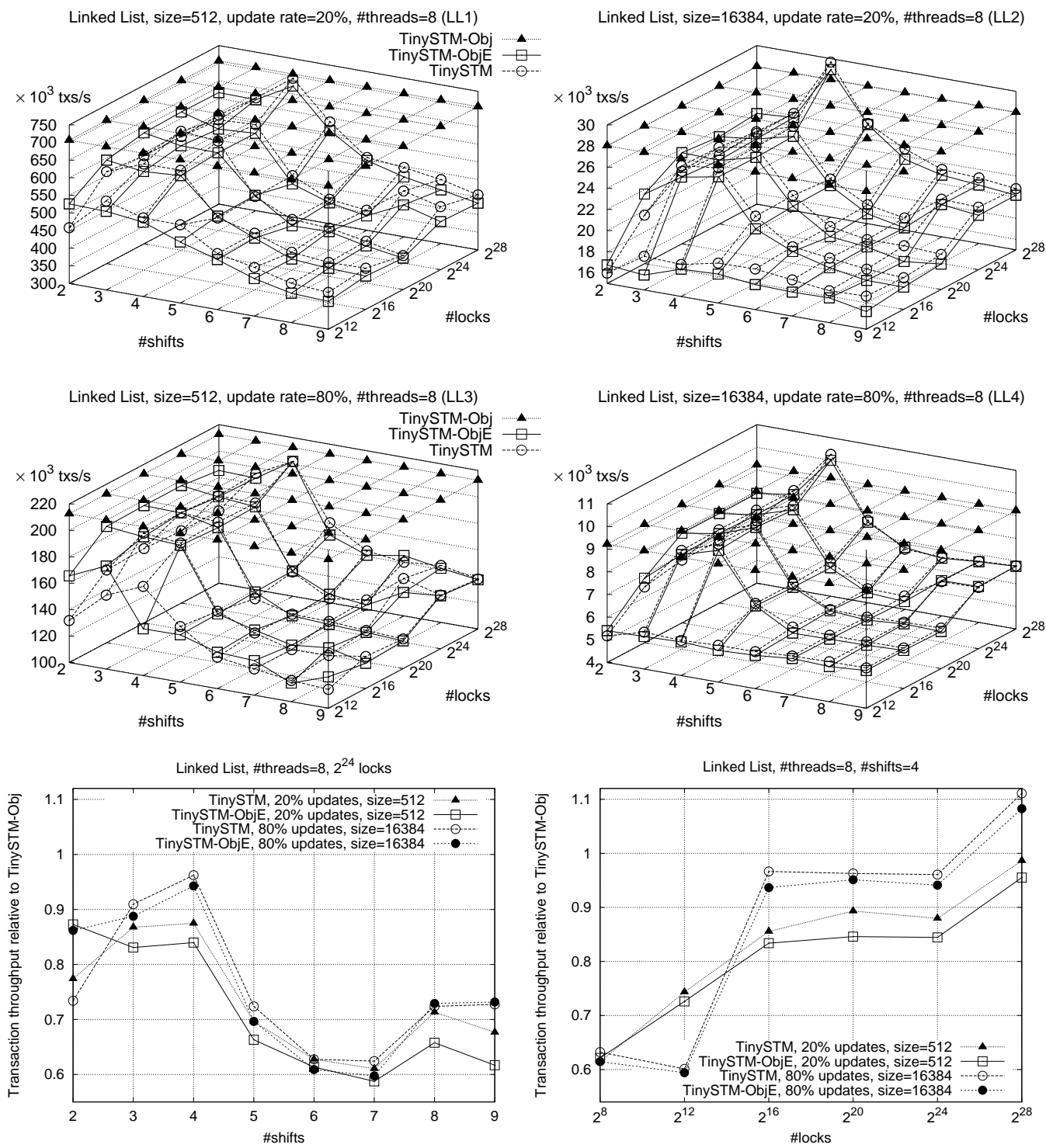


Figure 7. Transaction throughput for the Linked List benchmark.

Süßkraut, and Heiko Sturzrehm. Transactifying Applications using an Open Compiler Framework. In *TRANSACT*, August 2007.

[11] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory.

In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM Press.

[12] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict

Detection and Validation Strategies for Software Transactional Memory. In *20th Intl. Symp. on Distributed Computing (DISC)*, 2006.

- [13] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 78–88, New York, NY, USA, 2007. ACM Press.
- [14] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing Memory Transactions. In *PLDI '06: ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 14–25, New York, NY, USA, June 2006. ACM Press.
- [15] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The OpenTM Transactional Application Programming Interface. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 376–387, Washington, DC, USA, 2007. IEEE Computer Society.