Automatic Data Partitioning in Software Transactional Memories

Torvald Riegel
Dresden University of Technology, Germany
torvald.riegel@inf.tu-dresden.de

Christof Fetzer
Dresden University of Technology, Germany
christof.fetzer@tu-dresden.de

Pascal Felber
University of Neuchâtel, Switzerland
pascal.felber@unine.ch

ABSTRACT

We investigate to which extent data partitioning can help improve the performance of software transactional memory (STM). Our main idea is that the access patterns of the various data structures of an application might be sufficiently different so that it would be beneficial to tune the behavior of the STM for individual data partitions. We evaluate our approach using standard transactional memory benchmarks. We show that these applications contain partitions with different characteristics and, despite the runtime overhead introduced by partition tracking and dynamic tuning, that partitioning provides significant performance improvements.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Performance

1. INTRODUCTION

There exist a large variety of transactional memory designs (e.g., [8, 9, 7, 10]), each of which has different performance tradeoffs. Our experience with transactional memory indicates that there will be no "one size fits all" implementation. In particular, we expect that different workloads will require different optimizations and even different transactional memory designs. For example, one classical design decision in software transactional memory (STM) is the use of *visible* vs. *invisible* reads. The former makes readers visible to writers and hence, typically performs better than the latter on workloads with a high percentage of update transactions; it performs worse, however, for most other workloads. Another example is that of the granularity of conflict detection: memory regions that suffer from high contention might benefit from coarse-grained detection (e.g., at the object level, or even at the granularity of the whole region), while one would rather use fine-grained detection for non-contended regions.

One can expect that large applications will use different data structures that require different optimizations. For instance, a linked list might have a high update transaction rate and would benefit from visible reads, while a red/black tree in the same application with a low update transaction rate should rather use invisible reads. In reality, matters will likely be even worse: there might be multiple red/black trees, each of which might experience a different workload. This implies that optimizations of the transactional memory based on individual data types might not be sufficient to achieve good performance.

We believe that optimizing an STM for heterogeneous workloads is best addressed using a divide-and-conquer approach. Our key assumption is that the transaction workload is more homogeneous within rather than across "data partitions". Therefore, optimizations on a per-partition basis are more effective than optimizations on individual transactions alone and allow the STM to provide performance composability. For example, we could decide to use visible reads in highly contended partitions and invisible reads in partitions with low contention.

For our approach to be effective, we have to solve several problems. First, we need to find a good partitioning among all data structures of an application. Once partitions have been identified, we need to integrate the partitioning with the underlying STM and finally perform per-partition tuning.

Figure 1 illustrates our approach, which is based on a combination of compile-time and runtime techniques. We first automatically partitioning memory using the approach described in [6], thus allowing the STM to perform concurrency control separately for each partition. Second, we explain how STMs can be extended to use different kinds of concurrency control for different partitions, and how to tune the transactional memory system at runtime on a per-partition basis. In our current STM, tuning decisions are driven by runtime heuristics. We implemented our approach by extending Tanger [11] and TinySTM [10].

Using a hybrid compile-time/runtime approach allows us (1) to move most of the costs for establishing partitions to compile-time and (2) to still be able to support dynamically changing workloads (e.g., different workload phases) because tuning decisions and STM algorithms are chosen dynamically at runtime.

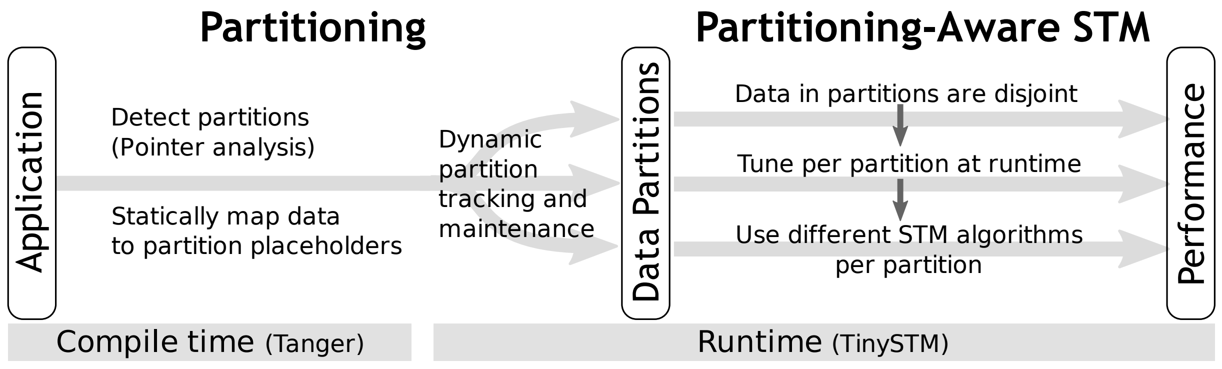

Partitioning
Partitioning-Aware STM
Application
Detect partitions (Pointer analysis)
Statically map data to partition placeholders
Dynamic partition tracking and maintenance
Data Partitions
Data in partitions are disjoint
Tune per partition at runtime
Use different STM algorithms per partition
Performance
Compile time (Tanger)
Runtime (TinySTM)

Figure 1: Our approach on a conceptual level.

We assume a model in which programmers only declare transaction boundaries and the compiler transforms memory accesses inside transactions into calls to the underlying STM. Several of the mechanisms proposed in this paper could also be introduced as manual optimizations, but this would be neither composable nor easy to achieve.

As we also aim to support transactions in systems software (which is often written in low-level languages like C), we specifically target unmanaged environments with pointers for which we cannot, for example, assume that memory is already partitioned into objects.

The remainder of the paper is organized as follows: We first describe related work in Section 2. We then describe the memory partitioning in Section 3 and introduce our partitioning-aware STM and its dynamic tuning approach in Section 4. We evaluate our approach in Section 5 and finally conclude in Section 6.

2. RELATED WORK

Partitioning is an intensely investigated topic in other areas such as distributed shared memories or resource scheduling. To the best of our knowledge, in the context of transactional memories there are only few works that use partitioning techniques. The most notable is the partitioning of workloads in [1] . Since the focus of this paper is to investige partitioning in the context of transactional memories, we do not discuss partitioning techniques in other areas.

Our goal is to reduce the cost of software transactional memory via partitioning. The underlying idea is that the access behavior of partitions can vary and, hence, we have the potential to optimize the cost on a per-partition basis. The major part of the overheads for uncontended transactions are per-access costs. For write accesses, these consist of undo/redo log maintenance, lock acquisition (for lock-based STMs), and write set maintenance. For read accesses, an STM can either use visible reads and acquire locks, or it can use invisible reads and perform validity checks. A read set/log is typically maintained by the STM, although this is not always necessary in time-based STMs [13, 7, 2].

The question of access granularity (e.g., word-based vs. object-based) can be seen as a partitioning problem. Although the decision for choosing one approach over the other has usually practical reasons (e.g., it may be difficult to determine the object that some arbitrary memory address belongs to), this of course also affects performance. For example, accessing large objects such as arrays with an object-based TM implementation could result in poor performance because of copying overheads or false sharing. Large objects could be split into smaller parts (i.e., partitioned) but this is far from trivial and we do not know of any implementation that currently does this.

Object-based TMs usually store transactional metadata in place, in the object itself. In word-based designs, one must be able to quickly locate the metadata associated with arbitrary memory addresses. To keep access overheads small, TMs usually use a simple hash function to associate metadata with memory addresses. This hash function splits all the memory into equally-sized blocks (e.g., machine words or cache lines) and associates them with entries in a preallocated metadata array. In all the TM implementations that we know of (except in [10]), this single hash function is defined globally for all accesses and cannot be changed at runtime. Obviously, optimizing such a fixed hash function for a specific workload may produce degraded performance for other workloads due to problems like false conflicts or too fine-granular locking [10].

Zilles and Rajwar use the birthday paradox to highlight the problem of false conflicts in hardware TMs [14]. They propose to use tagged ownership tables, which keep detailed ownership information to be able to distinguish false conflicts from real data conflicts. While this approaches fixes the problem once it has occurred, it would be better if we could remove its cause, i.e., the simplicity of the hash function.

Workload partitioning is proposed in [1] for dictionary-like data structures; a good partitioning will assign transactions to processors in such a way that the load is balanced and data locality is exploited.

In [12], Shpeisman *et al.* showed how an STM can use compile-time analysis and runtime mechanisms in a managed environment to track whether an object is thread-local or not accessed in transactions. This information is tracked per object, so memory is always partitioned into individual objects.

3. DATA PARTITIONING

In this section, we first give a high-level overview of our data partitioning approach before describing in detail its design and implementation.

3.1 Overview

We refer to functions that map entities from a given domain to a set of partitions as *partitioning functions.* For example, the hash function in word-based STMs that maps memory addresses to locks (or other metadata) is a partitioning function.

```
typedef struct node {
  intptr_t k, v;
  struct node *p, *l, *r;
  intptr_t c;
} node_t;

typedef struct rbtree {
  node_t* root;
} rbtree_t;

typedef struct manager {
  rbtree_t *car, *room, *flight, *customer;
  int nextCustomerId;
} manager_t;

void main(int argc, char **argv)
{
  manager_t* manager;
  // ...
}
```

Figure 2: Data structures in STAMP's *Vacation* application.

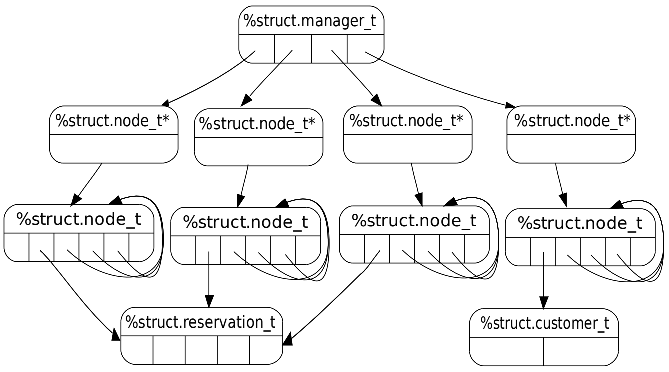


Figure 3: Simplified DS graph for STAMP's *Vacation* application. Each node represents one partition.

As an illustration of our data partitioning approach, consider the code in Listing 2 that shows a portion of STAMP's *Vacation* STM benchmark application [3] used in our evaluation (see Section 5). One can observe that the `manager_t` structure holds references to four different red/black trees. Each tree has a single root, as well as inner nodes with references to their parent and children.

Figure 3 shows a simplified Data Structure (DS) graph that was automatically inferred for the code shown in Listing 2. Informally, the nodes of a DS graph represent the memory partitions that a program creates and the edges of the DS graph specify to which partition (i.e., node in the DS graph) a pointer stored in a field within the partition can point to.

3.2 Identifying Partitions

To identify the partitions of an application, we first need to construct its DS graph. This is achieved using Lattner's Data Structure Analysis (DSA) [4], which is implemented as an analysis pass in the LLVM [5] compiler framework.

The DS graph is determined by analyzing where the pointers in a program are permitted to point to. The analysis is *field sensitive* in the sense that pointers stored in different fields of a data structure can point to different nodes (i.e., partitions) within the DS graph. However, a field can point to at most one node. To guarantee this property, whenever it is discovered that pointers stored in the same field point to disjoint nodes, these nodes are *unified.* This simple unification approach makes DSA fast and scalable.

The analysis is also *context-sensitive*, i.e., data structures are distinguished based on call graphs and not just allocation sites, for example. DSA can cope with calls to external functions, which implies that one does not need to analyze the complete program.

More precisely, our implementation is based on *Poolalloc* [6], which in turn uses DSA. Poolalloc operates as a transformation pass in the LLVM compiler framework. It construct a DS graph for every function encountered in the program. The graph lists the relationships between instances of data types used in the function. The information contained in DS graphs of callees of a function is merged with the graph of that function. A DS graph node is marked as *complete* in a function if DSA has analyzed all its uses. For example, a node is not complete if a pointer associated with it escapes to an external functions that has not been analyzed (i.e., it is passed as a parameter to an external function, returned by an external function, or accessible via an externally visible global variable). If two pointer values in a function are associated with different DS nodes marked as complete, then it is guaranteed that they point to non-overlapping memory regions. Otherwise, DSA would have unified the DS nodes to a single node.

Poolalloc then uses the DS graphs to create *pools* for complete DS nodes in each function and changes function calls and signatures such that callers pass pointers to pools that are required by callees. We can thus detect whether two pointers might address overlapping memory regions simply by comparing the pools associated with the pointers. Note that this relies both on compile-time and runtime information: the association between pointers and per-function pools is known at compile time, but callers of a function supply pointers to pools for incomplete DS nodes to callees at run-time and might pass the same pool for several arguments. Two separate data structures created using the same initialization function can thus be assigned to two different pools.

We can use Poolalloc's pools as partitions for the STM because distinct pools are guaranteed to contain non-overlapping memory regions. Looking at Figure 3, for example, an individual partition would be created for the sets of nodes in *each* red/black tree. Sets of nodes are not further partitioned because of unification.

Partition descriptors are instantiated at runtime when the control flow reaches a pool creation point in the program. These small data structures store the STM metadata for partitions (see Section 4). The STM's functions for transactional loads and stores receive a pointer to the partition descriptor of the target memory address. This information is used to handle memory accesses differently for each partition. If no particular partition is associated with the target address, a default partition is used.

3.3 Implementation Details

We have added data partitioning support to Tanger, our open-source STM compiler implementation [11] that also uses LLVM for "transactifying" C/C++ code. Therefore, the code analysis, data partitioning, and transactification phases are all integrated in the same tool chain. The code transformations target a typical word-based STM interface.

The original motivation for Poolalloc was to improve performance by increasing the locality of heap objects belonging to one data structure. To achieve that goal, Poolalloc uses a custom per-pool memory allocator. In our implementation, we do not use custom allocators as this would add another dimension to the problem space and would exceed the focus of this paper.

We had to perform several modifications to Poolalloc and DSA to adapt them to STM partitioning. Most of these modifications are related to multi-threading, e.g., we need to allocate pool descriptors on the heap rather than on the stack as they are shared between threads, we added garbage collection for pool descriptors since they are now allocated on the heap, we have to handle pool descriptors that are passed as parameters to functions that create new threads, etc. Our compiler can also detect transaction-local partitions (i.e., partitions which are created and destroyed in the same transaction) and thread-local partitions in simple cases.

Our current implementation creates one pool/partition for each complete node of the DS graph, although it would be possible to group several nodes in the same partition. Coming back to the example of Figure 3, it might make sense to use one partition per red/black tree, i.e., one might want to collapse the root and inner nodes of each tree.

We also do not partition data structures for which Poolalloc uses only a single node in the DS graph, but we could of course sub-partition memory in a Poolalloc-based partition using alternate mechanisms.

4. PARTITIONING-AWARE STM AND DYNAMIC TUNING

Our main objectives with partitioning are (1) to isolate the partitions with respect to concurrency control and (2) to facilitate the tuning of the STM by considering each partition independently. The underlying idea is that a typical application will have a variety of data structures, each of which has different transactional workloads. Hence, by selecting a concurrency control mechanism suitable for each partition and by tuning each partition individually, we can improve the overall transactional throughput.

For our prototype implementation, we use TinySTM [10], a word-based STM design that uses versioned locks to protect shared memory locations and invisible reads with time-based validation [13] (please refer to [10, 7, 2] for details on the operation of such an STM). It relies upon a shared array of locks to manage concurrent accesses to memory. To obtain the lock associated with a memory address, the STM first shifts off a certain number of least-significant bits and then masks the most-significant bits: $lock = (addr >> \#shifts) \bmod \#locks$. The number of locks is a power of 2 and the number of shifts determines how many contiguous memory locations are protected by a single lock. Both parameters have a large influence on the performance of an STM and the optimal setting depends on the workload [10].

The original STM implementation is not aware of partitions (it uses a single partition) and has to choose one configuration for controlling concurrency in the whole application. Also, all transactions have to use the same global lock array, which can result in false conflicts and makes tuning more difficult.

Using multiple partitions allows us to perform various kinds of optimizations that would be ineffective with a single partition. For example, it is unreasonable to assume that a single global partition would be read-only, but it is not unlikely that some partitions in an application would be read-only (see Section 5).

Table 1 shows the types of concurrency control for partitions that our partitioning-aware STM currently supports. They offer various tradeoffs in terms of concurrency and overhead. Note that even though the STM can use a different algorithm in each partition, it still provides transactions with the same guarantees as the original STM does. The integration of other concurrency control mechanisms than those shown in Table 1 should be straightforward because partitions are guaranteed to not overlap.

When a partition is created at runtime, the STM stores metadata for the partition in the partition descriptor, which is passed by reference upon every access to the partition (see Section 3). The descriptor structure consists of fields for (1) the partition's type (see Table 1), (2) a single lock for the Shared Lock and Exclusive Lock types, (3) a pointer to the lock array, the number of locks, and the number of shifts for the Multiple Locks type, and (4) a few counters to maintain statistics (e.g., the number of aborts in the partition).

The STM also provides a default partition for every transactional memory access that is not associated with a partition. This is the fall-back implementation for accesses that, for example, target incomplete DS nodes (e.g., data that is also accessed in external functions, see Section 3) or accesses that are assumed to be associated with a partition by a callee but not by the caller. The default partition is of type "Multiple Locks".

On each transactional memory access, the STM loads the type of the partition from its descriptor and dispatches execution to the code responsible for this type. This constitutes a large part of the runtime overhead of partitioning (see Section 5) but much of this overhead can be removed by better compiler optimizations (e.g., creating different code paths specialized for different partition types).

Partitions can be tuned on demand and independently of each other. When a thread wants to tune a partition, it (1) tries to change the partition type to "Tuning" using a compare-and-set instruction, (2) acquires a new timestamp from the global clock used by the STM for time-based validation [13], and (3) waits until every active transaction has a start timestamp larger or equal than the acquired timestamp. If a transaction accesses a partition that is being tuned, it aborts and updates its start timestamp. Thus, if step (1) succeeded, then after step (3) every transaction will discover or will already know that the partition is being tuned. The thread that performs the tuning can thus change the partition's metadata and finally set the new partition type. Note that, although we do have to wait for active transactions to complete, tuning will only delay the transactions that actually access the partition being tuned.

Our current prototype uses simple tuning strategies (see Table 2) based on runtime measurements and heuristics. All strategies initially set the partitions to "Read-Only". The type of a partition is changed if the number of aborts in the partition exceeds a certain threshold (e.g., on reaching 1,000 aborts, Part-3 changes the partition type to "Mul-

Type	Concurrency control	Performance	Purpose
Multiple Locks	Per-partition array of locks. Similar to using one instance of the original STM per partition.	The indirection via partitions adds some overhead over the original STM.	General purpose.
Shared Lock	Single per-partition lock embedded in partition descriptors. Same algorithm as with multiple locks.	Lower overhead than "Multiple Locks" but supports only a single updating transaction.	Mostly-read and uncontended partitions.
Exclusive Lock	Single per-partition exclusive lock for both reads and writes.	Does not allow concurrent accesses by transactions. Lower overhead than "Shared Lock" because reads do not need to be validated.	Partitions that are rarely accessed concurrently at runtime.
Read-Only	No concurrency control. Does not allow updates.	Very low overhead but the partition type must be changed when a transaction wants to update data.	Read-only partitions.
Tuning	Transactions will abort.	N/A	Tuning.
Thread-local	No concurrency control. Undo-logging for writes.	Low overhead.	Special purpose.
Transaction-local	No concurrency control.	Very low overhead.	Special purpose.

Table 1: Partition types supported in our implementation.

Strategy	First Update	20 aborts	1000 aborts	2000 aborts
Part-1	Multiple (L=2^{18}, S=6)			
Part-2	Single Exclusive	Multiple (L=2^{18}, S=6)		
Part-3	Single Exclusive	Single Shared	Multiple (L=2^{18}, S=6)	
Part-4	Single Exclusive	Single Shared	Multiple (L=2^{10}, S=8)	Multiple (L=2^{18}, S=6)

Table 2: Partitioning Strategies. Initially, all partitions are of the "Read-Only" type.

tiple Locks", an array of 2^{18} locks, and a 6-bit shift). We chose values for the number of locks and shifts that provided good overall performance in our benchmarks (see Section 5). Please note that while our approach supports workloads that dynamically change their characteristics at runtime (e.g., initializing a lookup table and later using it just for read-only lookups), simple tuning strategies can effectively limit the ability of the STM to adapt to these changes. For instance, our current strategies never tune a partition back to the Read-Only type.

5. EVALUATION

To evaluate the effectiveness of our approach, we have used the Vacation, Genome, and KMeans benchmarks from the STAMP [3] TM benchmark suite. In these benchmarks, memory accesses are only transactional if specially marked. However, we assume that programmers would rather rely on compilers to "transactify" applications and that programmers will not tell compilers which accesses can be safely executed non-transactionally. We therefore used Tanger [11] to compile all benchmarks. We produced versions with and without partition support. Applications were compiled to 32-bit executables and run on an 8-core machine[1]. We also used the classical *Linked List* micro-benchmark used to evaluate most STM designs for illustrative purposes.

The first stage of our approach takes place during compilation of applications. Table 3 shows how many of the transactional loads and stores in an application are associated with partitions. Accesses not associated with a partition are implicitly linked to the default partition by the STM (see Section 4). Partition creation points are calls in the program code that instantiate partitions. Note that the number of partitions actually created and accessed at runtime can be different from the number of creation points. The table shows that most of the accesses can be associated with partitions. KMeans has fewer partitioned accesses because it uses global variables, for which Poolalloc does not by default create partitions.

Table 4 shows runtime statistics for partition accesses and aborts. All partitions were forced to be of the "Multiple Locks" type and to use 2^{18} locks and 6 shifts. Partitions without transactional accesses have been omitted. For load-/store statistics, we used a single thread, default parameters and 2M transactions for Vacation, low contention settings and a 2K input file for KMeans, and 40K segments[2] for Genome. For abort statistics, to produce contention we used

Benchmark	Partition creation points	Partitioned/ total transactional loads	Partitioned/ total transactional stores
Vacation	19	160 / 164	103 / 105
KMeans	11	5 / 8	1 / 4
Genome	23	47 / 47	15 / 16
Linked List	3	13 / 13	6 / 6

Table 3: Compiler statistics.

[1]Two-way quad-core Intel Xeon at 2 GHz running Linux 2.6.18-4 (64-bit)

[2]`-g16384 -s64 -n41943`

Benchmark / Partition	Txnal Loads	Txnal Stores	R/W aborts	W/W aborts
Vacation / 1	105M	11M	1.3K	43K
2	125M	20M	2.8K	33K
3	4.3M	450K	9	3K
4	192M	3.5M	107K	10K
5	253M	184K	192	0
6	258M	41K	33	0
7	37M	15M	17K	63
8	263M	41K	24	0
9	6.9M	5	328	0
10	8.2M	0	0	0
11	8.4M	0	0	0
12	8.4M	0	0	0
13	22M	0	0	0
default	21M	19M	5K	894
thread-local	14M	10M	0	0
KMeans / 1	524K	524K	451K	28K
2	524K	0	0	0
3	524K	0	0	0
4	524K	0	0	0
5	32K	0	0	0
default	44K	44K	16M	157K
Genome / 1	11.8M	106K	316	49
2	91K	30K	186	436
3	118M	4.7M	567K	3K
4	952K	0	0	0
5	65K	75K	0	14
6	15K	0	0	0
7	1.9M	0	0	0
8	952K	0	0	0
9	30K	0	0	0
10	0	15K	0	0
11	337K	75K	1K	516
12	57K	0	0	0
13	141K	0	0	0
14	42K	0	0	0
15	84K	0	0	0
txn-local	73K	0	0	0
thread-local	0	42K	0	0

Table 4: Runtime statistics for partitions: transactional accesses and aborts due to read/write or write/write conflicts.

8 threads, a 64K input file for KMeans, and 4M segments[3] for Genome.

We can see that the number of accesses varies a lot between the partitions. There are more reads than writes, but the relation differs per partition. There are several read-only partitions but the largest partitions are often updated (e.g., in Vacation). The default partition receives much less accesses than the other partitions or none at all (Genome). This number could be decreased by tuning Poolalloc's heuristics for when to create partitions, and by improving the thread-local compiler analysis. Transaction abort counts have also a high variance, which further shows that partitions are different. Note that we do not use any kind of contention management. We did experiment with different backoff schemes but results were not conclusive. Overall, Table 4 shows that data partitioning creates many opportunities for different kinds of optimizations.

Table 5 illustrates the performance of the different partition types (see Table 1 for details about each type). We ran the benchmarks with a single thread and forced all partitions to be of a certain type. The Linked List benchmarks run transactions that look for a specific element in lists with 2000 and 250 elements, respectively. In our current STM prototype, partitioning adds non-negligible overhead to transactional accesses. The first reason is that the STM dispatches execution based on the partition type for every access (see the difference between the second and the third column). Further compiler optimizations could remove this overhead. For example, the compiler could detect that only one partition is used in a function and create a special version optimized for read-only or thread-local partitions.

The second part of the overhead (third column) is due to the extra level of indirection that partitions represent. The partitioning-aware STM has to load the pointer to the lock array and the number of locks and shifts from the partition descriptor, whereas these values are fixed in the original STM. Nevertheless, the other columns show that despite these overheads, even just using a single lock instead of multiple locks can increase performance significantly. Further compiler optimizations should also increase the performance advantage of the partition types that need no or very little per-access concurrency control code (e.g., "Read-Only" or "Exclusive Lock") because no calls into an STM library are necessary and undo-logging can be efficiently inlined in the application code.

After showing the applicability of partitioning and the potential of optimizations that it enables, we now show performance results for the benchmarks from the STAMP suite. Figures 4, 5, and 6 each show the performance of configurations of the original STM and of the four simple tuning strategies of the partitioning-aware STM shown in Table 2. Performance results are shown as speedup relative to the single-thread performance for the first configuration of the original STM. The lock/shift settings that we use for the original STM (labeled `Orig-L-S` with 2^L being the number of locks and S the number of shifts) were the ones that provided the best performance for eight threads.

For Vacation (Figure 4), the partitioning-aware STM is often slightly slower than the original STM due to the higher per-access overhead. However, it performs significantly better if (1) it uses the "Exclusive Lock" type in single-threaded runs or (2) in the high contention variant of the benchmark with a large number of threads. The latter also shows that increasing the number of locks in the original STM (`Orig-24-6`) is not sufficient to avoid false conflicts in a single lock array; in contrast, the partitioning-aware STM can still scale even though it uses less locks.

In KMeans (Figure 4), the partitioning aware STM can take advantage of the three often-accessed read-only partitions (see Table 4). Using fine-granular locks seems to be important in the high-contention variant of the benchmark, which shows that advanced tuning strategies should also adapt the number of locks and shifts for each partition.

In Genome (Figure 6), the partitioning-aware STM performs significantly better than the original STM (with one exception). Although the statistics in Table 4 suggest otherwise, we observed that the two most-frequently accessed partitions (1 and 3) are read-only during the first phase of

[3] `-g16384 -s64 -n4194304`

Benchmark	Multipe Locks	Multiple Locks, No Dispatch	Shared Lock	Exclusive Lock	Read-Only
Vacation	0.71	0.80	1.21	1.32	N/A
KMeans	0.85	0.94	1.07	1.13	N/A
Genome	0.74	0.84	1.17	1.34	N/A
Linked List Large	0.68	0.86	1.36	1.90	2.30
Linked List Small	0.61	0.80	1.29	1.73	2.22

Table 5: Performance of partition types and overhead of supporting partitions. Shows speedup relative to the original STM.

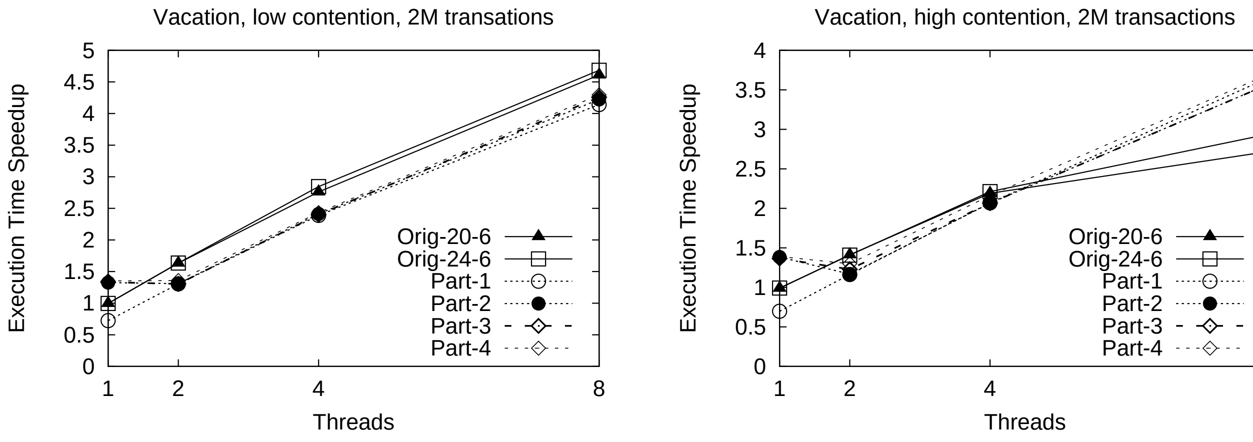


Figure 4: Results for Vacation's low and high contention variants.

the benchmark, which allows the STM to handle many of the accesses using a "Read-Only" partition type. More importantly, the right part of Figure 6 shows that partitions can decrease the STM's space overhead tremendously. `Part-4` uses five partitions with 256K locks and one with 1K locks; the original STM uses 256MB of memory just for its locks and it was not able to execute the benchmark with 2^{24} locks in a reasonable amount of time due to false conflicts.

6. CONCLUSION

The runtime overheads of current STMs are not negligible and there is still much room for optimizations. We believe that partitioning will play a central role in optimizing STMs for real-world workloads. Partitioning enables STMs to compose different specialized optimizations that would not be beneficial globally, for example when dealing with uncontended or read-only partitions.

We showed how to add partitioning support to an STM compiler and runtime system. Our approach is applicable to software written in low-level languages such as C and does not require managed environments. The performance results for several STM benchmarks showed that even our non-optimized prototype can already yield better performance than the original STM without partitioning support, often with a much lower memory overhead.

Furthermore, we believe that partitioning support enables many other improvements for TM beyond optimizing STMs. For example, hybrid TMs could choose to use hardware TM (HTM) only for contended partitions and thus be able to overcome limitations of a particular HTM implementation. Using partitioning to reduce false conflicts should be applicable to HTMs too. Finally, TM building blocks such as schedulers, contention managers, or workload analyzers should benefit from the additional level of information provided by partitioning.

Acknowledgements.
We are very grateful to Andrew Lenharth and Vikram Adve for kindly answering our questions about DSA and to Nir Shavit for his constructive feedback.

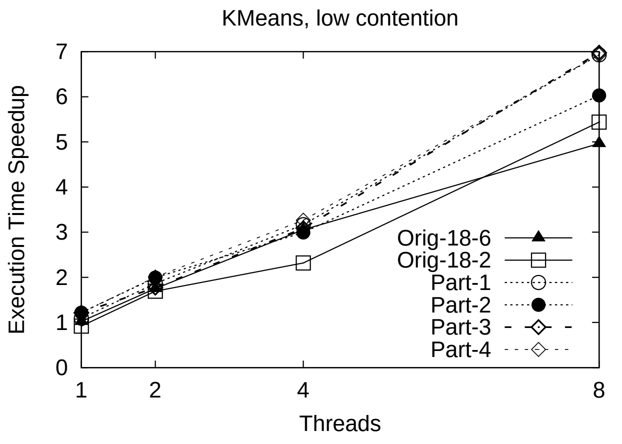
7. REFERENCES

[1] T. Bai, X. Shen, C. Zhang, W. N. Scherer III, C. Ding, and M. L. Scott. A key-based adaptive transactional memory executor. Technical Report TR 909, Computer Science Department, University of Rochester, Dec. 2006.

[2] 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.

[3] 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.

[4] 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`.



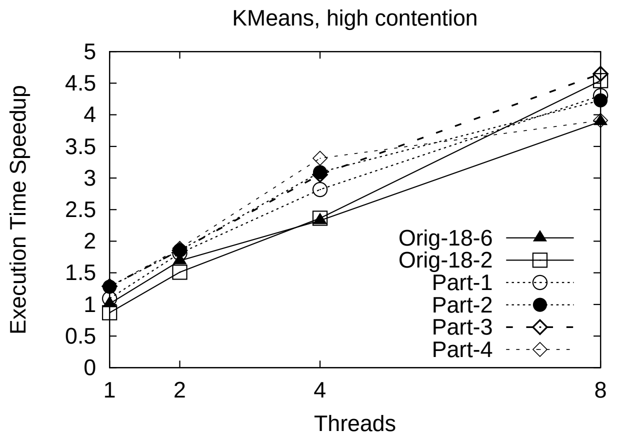
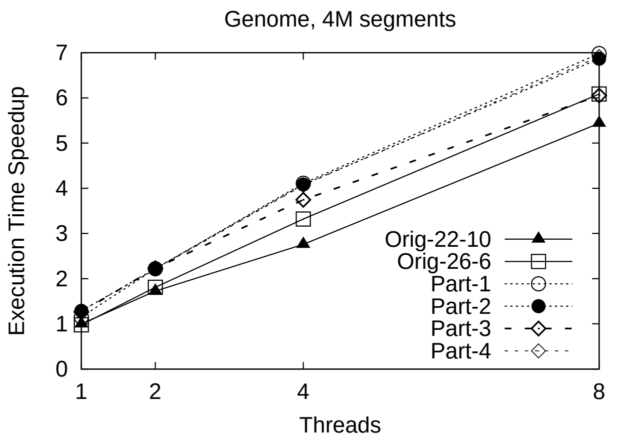


Figure 5: Results for KMeans's low and high contention variants.



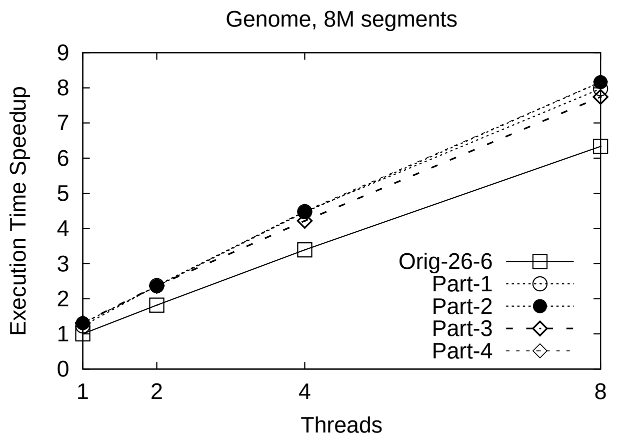


Figure 6: Results for Genome for workloads with four and eight million segments.

[5] 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.

[6] Chris Lattner and Vikram Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, Chigago, Illinois, June 2005.

[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.

[8] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of OOPSLA*, pages 388–402, Oct 2003.

[9] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[10] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[11] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying Applications using an Open Compiler Framework. In *TRANSACT*, August 2007.

[12] 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.

[13] Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *20th International Symposium on Distributed Computing (DISC)*, September 2006.

[14] C. Zilles and R. Rajwar. Brief Announcement: Transactional Memory and the Birthday Paradox. In *19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2007.