DREAM: Dresden Streaming Transactional Memory Benchmark

Jons-Tobias Wamhoff Stefan Weigert
Technische Universität Dresden, Germany
first.last@tu-dresden.de

Abstract

Transactional memory is a concurrency control mechanism that aims to simplify the development of parallel applications in the times of ever-increasing core counts of multi-processor architectures. At the same time, it must support the application to achieve a high level of performance and scalability to justify it as an alternative to classical lock-based synchronization. These claims still lack a broad validation in the field because the high number of proposed transactional memory algorithms are evaluated using only a small variety of existing benchmark applications.

In this paper, we introduce DREAM, an extensible benchmark framework for processing live updates to large graphs using transactional memory. It represents real-world workloads from the domain of data stream processing. Popular social networks, like Facebook, Twitter or Google+ have increased the economical interest in computing on large graphs. However, social networks are *scale-free* - i.e. their degree distribution follows a *power-law*: The majority of edges connect to the minority of vertices, which makes computations on these networks hard to scale. On the other hand, "Big Graphs" don't necessarily mean "Big Data". Even huge networks can be processed efficiently on a single machine, especially when done in parallel.

We currently support in the DREAM framework two operators that implement standard graph algorithms which operate on streamed updates: community of interest and community detection based on label propagation. However, DREAM is designed very modular and new algorithms can be added straightforward by implementing them as new operators. We evaluate the use of DREAM with well-known software transactional memory implementations.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.8 [*Software Engineering*]: Metrics–performance measures

General Terms Measurement, Performance, Experimentation.

Keywords Benchmarks, Transactional Memory, Concurrent Programming.

1. Introduction

System with processors that have multiples cores are now the defacto standard, even in embedded systems. These systems provide the applications the potential for significant performance improvements by exposing parallelism to the system, e.g., by spawing multiple threads. However, the development of parallel applications is a difficult task. Their behavior must remain the correctness of the sequential application while being efficient in parallel execution. The development of multi-threaded applications requires a high effort for the synchronization of data accesses. Concurrency control is typically achieved by augmenting critical sections with locks. Finding the right balance for the lock granularity is an expert domain.

Transaction Memory (TM) [16] is a concurrency control mechanism that hides the hassle of implementing the synchronization of applications away from developers. All data accesses within the demarcations of a transaction are redirected to the TM. The TM transparently introduces the synchronization from its underlying algorithm to the application. Transactions atomically commit their changes if no access conflicts were detected and the data consistency is guaranteed. The parallel execution of transactions is isolated from each other. In case of a conflict, the transaction is aborted and all changes are reverted so they do not become visible externally.

A large number of algorithms has been proposed in the recent years that implement TM either in software (STM) [9, 11, 13, 18, 29], hardware (HTM) [6, 17, 25] or hybrid in software and hardware [10, 28]. The evaluation of these designs is based on a relatively small number of benchmark applications that specifically target TM [2, 5, 15, 20]. The existing benchmarks represent only a subset of all application domains and workload types that are relevant for TM. Our goal is to increase the variety of possible workloads. Therefore, we propose a new benchmark framework, called *Dresden Streaming Transactional Memory Benchmark (*DREAM*)*, that allows a straightforward implementation of typical workloads from the domain of data stream processing [1, 19]. DREAM simulates a single node of the stream processing system. The stream processing node is modeled as a thread-pool that processes operators. The operators are stored in queues and a scheduler assigns them dynamically to the threads for execution. The actual workload is implemented using one or multiple operators that execute on the node.

Popular social networks, like Facebook, Twitter or Google+ have increased the economical interest in computing on large graphs. In social networks, the fraction of individuals that have just a few connections is significantly higher than the fraction of individuals that have thousands or millions of connections. It is easy to see that this is true in practice: Lady Gaga, for example, currently has $32,099,427$ followers on Twitter, while one of the Authors has 8 followers only. Obviously, there are only a few accounts like Lady Gaga's on Twitter, but there are many "ordinary" accounts with only a few followers. This property is called *scale-free* and it has been shown that this is an important property of every social network [24]. It defines that the degree distribution in a graph follows a *power-law*. The degree of a vertex equals the number of connections it has.

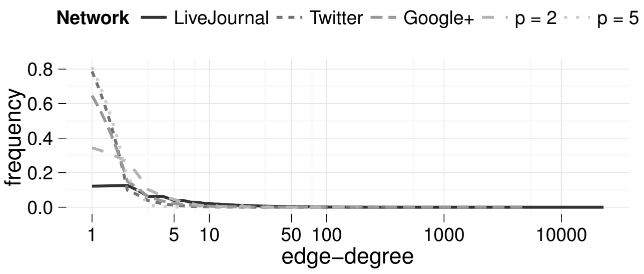

Figure 1. Distribution of the edge-degrees in three real social networks and two power-law distributions (p).

Figure 1 shows the degree distribution for three social networks. The data is freely available through the *Stanford Large Network Dataset Collection (SNAP)*[1]. Running parallel algorithms on top of these graphs is difficult [14, 21] because it is hard to distribute the load evenly for a good scalability. At the same time it has been shown that such "Big Graphs" do not equal "Big Data" and even huge networks can be processed efficiently on a single machine [21, 30]. We believe that this poses an interesting new challenge on transactional memory implementations since contention will not be distributed equally.

We believe that TM can play a key-role in helping statisticians and others to run graph algorithms on large graphs efficiently: It offers an easy-to-understand language construct which requires no expert-knowledge when parallelizing sequential algorithms. However, the *scale-free* property of the input challenges the implementations of TM because the graph algorithms will have to access (through reads and writes) a small fraction of the vertices significantly more often than the majority of the vertices.

We developed two different operators using DREAM to expose scale-free workloads from the domain of data stream processing. The operators currently support two standard graph algorithms which operate on streamed updates: community of interest [7] and community detection based on label-propagation [26]. New algorithms can be added easily due to the modular design by implementing them as operators in the processing model.

The rest of the paper is structured as follows: We introduce the DREAM benchmark design and the implementation of the two graph algorithms in Section 2. Section 3 contains an extensive set of experiments which show the behavior of DREAM for current TM implementations. The relation to existing TM benchmark applications is discussed in Section 4. Finally, we conclude this paper in Section 5

2. DREAM Benchmark

Our goal is to provide a platform for the simulation of data stream processing operators. Multiple operators reside on a single node and are executed concurrently. The operators are stateful and the state is shared among all operators. We focus on the maximum utilization of the computing power of the node's processors in conjunction with transactional memory (TM).

The overall design is based on the following components:

[1] `http://snap.stanford.edu/data/index.html#socnets`

- The data stream is processed by *operators*, which implement the workload. Different kinds of operators exist that can be used to implement for or while loops, thread barriers, exclusive operations, etc. If the operators do not execute exclusively, they can use TM to access the state.
- The operators are stored in a global *queue* until they are executed. A scheduler dispatches the operators from the queue and maintains their properties, e.g., an exclusive operation.
- A *thread-pool* of configurable size consists of the threads that execute the operators that they get assigned by the scheduler.
- The *data stream* is located already in memory in our simulation. It can be raw data that is generated randomly or read from files, or preprocessed data, e.g., partitioned into batches. Reading the data into memory can be implemented by special operators.

The actual workload that should be executed by DREAM is implemented using one or multiple operators. The parallelization is handled by the framework transparently when it maps the operators onto the threads. Thus, combining the operators with TM for synchronization is very beneficial because it handles the synchronization transparently. The operators are implemented as a function in C++ and can be dispatched to the queue by assigning their function pointer to a job kind. The DREAM framework will call the function when it gets scheduled, depending on the job kind, one or multiple times and in isolation or in parallel.

We implemented two kinds of operators with standard graph algorithms to compute communities of interest (see Section 2.1) and community detection based on label propagation (see Section 2.2).

The operators contain transactions that protect the access to the shared state. The instrumentation of the code is performed automatically using the Dresden TM Compiler (DTMC) [6]. The underlying interface to the TM allows to plug in a large selection of TM implementations for the evaluation in Section 3.

2.1 Communities Of Interest

A basic observation in social networks is that individuals have a "social fingerprint": Their most important contacts. A consequence of this observation is, that it is not necessary to store all the edges of a vertex - just the ones that are important in the actual context. *Communities of interest (COI)* [7] use this observation. COI applies a top-k algorithm to the original graph to transform it into a COI graph with a limit k of edges per node. For example, one can use the length of phone-conversations to determine which contacts are important or not [7]. In our previous work we used the amount of bytes transferred between computers in the internet to determine the importance of TCP-connections [32].

This can be very useful, since it (1) limits the amount of memory that is needed to store a graph by limiting the amount of edges in the graph and (2) permits us to tune the skew of the degree distribution precisely. If $k = \infty$, then any algorithm, running on top of the COI-graph, is subject to the scalability limitation, induced by the power-law. In fact, in the case where $k = \infty$, the COI-graph equals the original graph. If $k = 1$, then the graph is not scale-free anymore and scalability is not limited by the nature of the input. Any $k > 1$ will increase the scale-free characteristic of the input.

We have already shown that COI can be computed on streaming data [32]. The idea is to record a fixed-size window of input for each vertex (see Figure 2). Once the window is full, the elements, contained in the window, are merged with the COI of the corresponding vertex. This procedure is re-executed as long as there is more input to process.

The COI operator in DREAM takes as input a window of maximal size w. The window can either be preprocessed data read from a file and sorted by the source ID or gets generated randomly

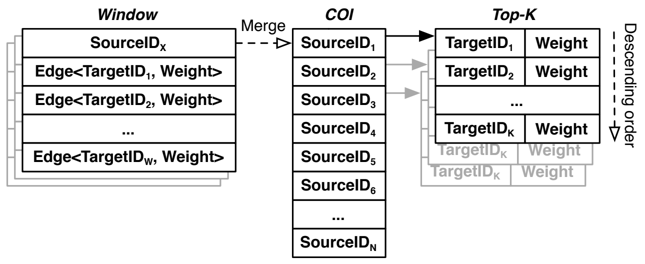

Figure 2. Illustration of the COI algorithm: The windows are merged into the top-k entries that are ordered by a descending weight.

according to the power law distribution. The operator finds the top-k list that corresponds to the source ID using the COI table. It then merges the entries of the window into the top-k list, which is sorted in descending order by the weight. The weight is calculated as an moving average based on the factor θ (see Section 2.3). Updating the top-k is protected by transactions that either change the weight or replace entire entries in the list.

2.2 Community Detection

Community detection in graphs can be done in several different ways. One option is called *label propagation (LPA)* [26].

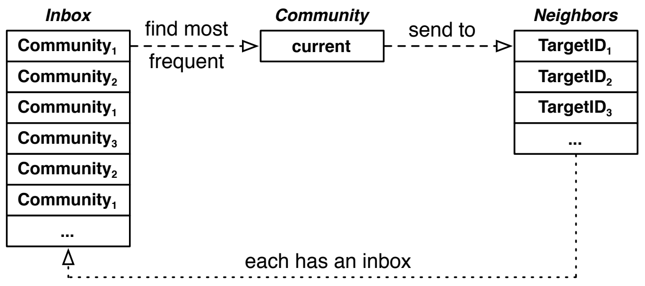

Figure 3. Illustration of the LPA algorithm: A vertex decides on the most frequent community and sends it to all its neighbor's inboxes.

This algorithm is executed in several iterations (see Figure 3) and each step is implemented as an operator. On the first iteration, every vertex in the graph chooses a unique community. This could be, for example, initially its own id. Every vertex will then "send" its own community to all its neighbors. This operator uses transactions to synchronize the append operation on the inbox. On every subsequent iteration, each vertex counts the different communities it received from it's neighbors in the previous iteration and selects the community with the highest count as its current community. If there is no clear winner in the counts, the vertices will select a random label. The decision on a community is a private operation and does not need any synchronization.

The idea is that densely connected clusters (i.e., communities) of the graph will converge to a consensus eventually. The algorithm terminates if, during the current iteration, no vertex had to change its label.

2.3 Workload Characterization

The scale-free concurrency pattern of social graphs provides an interesting workload for TM. It can be customized by a wide range of configuration parameters that are summarized in Table 1. We currently do not evaluate the feature set of TM, such as privatization or irrevocability.

Parameter	Description
`num-threads`	Number of threads in the thread-pool
`phases`	Select the executed operator(s): COI, LPA or both
`window-size`	Size of the time window that will be merged by the COI operator
`topk-limit`	Number of top-k entries maintained by the COI operator
`theta`	Defines how much a window entry influences the weight in the top-k during a merge: $\theta * weight_{top-k} + (1-\theta) * weight_{window}$
`file-input`	Select input streaming data from file (accessed randomly or in chunks) or generate windows randomly
`file-name`	Input file with the edges of the graphs in format $< source\ target\ weight >$ (default weight is 1 if not specified)
`duration`	Runtime duration in milliseconds for the COI operator when using random windows
`number-max`	Largest ID for source and target for random window generation
`power-law`	Power law distribution of the targets in windows generated randomly
`seed`	Seed for the random number generator
`lpa-iterations`	The number of iterations after which the LPA operator will be stopped if it did not terminate earlier

Table 1. Overview of the configuration parameters for COI and LPA operators.

The workload's concurrency pattern is summarized in Table 2 and characterized as follows:

- The *transaction length* covers the total number of accesses to memory, either read or written.
- The *write-set size* represents the number of updates to memory and indicates if a transaction is read-only.
- The *time spent in transactions* characterizes the impact of TM on the overall scalability of the operator.
- The level of *contention* shows the conflict probability that will lead to an abort of a transaction.

With the DREAM benchmark, a wide range of concurrency patterns can already be covered by the presented operators and adjusting their parameters accordingly. The introduction of a power law distribution into the workload can shift the level of contention from a read-only pattern to a write dominated complex access pattern.

The benchmark also allows the verification of the computed result. This is an important property for the comparison of the computed results of underlying TM implementations with a sequential golden run. For the COI operator, the verification is a comparison of the top-k entries and for the LPA operator the computed communities are compared.

3. Evaluation

We evaluate the DREAM benchmark using the following selection of current software transactional memory (STM) implementations: Two STM variants that are based on the lazy snapshot algorithm [27]: TinySTM [12] is based on a versioned array of

Concurrency Pattern	**COI operator**	**LPA operator**
Transaction description	Merges an entire window into the top-k list of a single source ID	Sends the current community of a single source ID to all its neighbor's inboxes. The neighbors are derived from the computed top-k list of the COI operator.
Transaction length	Directly results from the number of entries in the window (`window-size`) that are merged with the top-k list (`topk-limit`). The maximum length is reached if `window-size` entries have to be matched against all `topk-limit` entries. In the minimal case, the `window-size` entries are matching the first top-k entry.	The transaction length only depends on the `topk-limit`. The transaction performs a lookup of the community and performs an update of the target's inbox.
Write-set size	Only a few memory locations are modified when updating the top-k list. The minimal size is a single update of the weight. The maximum size results from a reordering of the top-k list, which includes the removal from the old position and an insertion at the position that corresponds to the descending order.	The operator is write dominated. The size of the write-set is equal to the number of entries in the top-k list (`topk-limit`) because for each entry the community is appended to the inbox.
Time spent in transactions	Almost all time of the operator is spent in transactions. Only the random generation of windows is a private operation.	Only sending the community is performed in transactions. Computing the current community is a local operation on partitioned data and requires no synchronization.
Contention	It depends on the `power-law` distribution, the number of target IDs `number-max` and the the `topk-limit`. A high power-law setting will bound the majority of edges to a minority of targets. This results in a shorter period of recurrences of targets in the windows and thus a higher probability of updates on the top-k list that can result in a conflict. The overall level of contention is low because the operator performs only few update operations.	It is derived from the `power-law` distribution and the `topk-limit` of targets. A high contention level is reached when the top-k entries are bound to a small number of neighbors due to the power-law. The inboxes of these neighbors will suffer from high level of contention when all sources add their current community.

Table 2. Overview of the concurrency patterns for COI and LPA operators.

locks and operates either in write-through mode (WT), i.e., directly updating to memory, or in write-back mode with encounter time locking (ETL), i.e., buffering updates with eager conflict detection; and two STMs are based on a single versioned lock, either exclusively directly updating memory (TML [8]) or performing buffered updates and value-based validation (NOREC [9]). FASTLANE (FL) [31] uses a combination of pessimistic transactions that run almost at native speed and speculative transactions that synchronize using a lock array.

We used data-sets to simulate the stream that fit the properties of graphs in social networks at presented in Section 1. The windows of `window-size` (W) is generated randomly. The `power-law` (P) parameter allows to adjust the distribution among the `number-max` (N) node IDs. The COI operator maintains a top-k list of `topk-limit` (K) entries. While the generated data imitates data, the DREAM framework is also able to read the data from input files that capture real data from various networks. The Stanford Large Network Dataset Collection[2] offers a wide range of data from social and other networks. We chose to generate the input data randomly because it has similar properties as the real data (see Figure 1) but allows to execute the application for user defined durations.

We evaluate the following two scenarios: First, the data is fed into the COI algorithm, which constructs a COI graph (see Section 3.1). This simulates a monitoring of a social network, where the graph does not change but updates are received at a high frequency. Second, the LPA algorithm runs on the COI graph (see Section 3.2). This scheme could be used, for example, to detect attackers who create malicious or fake identities in social networks to gain influence (e.g. click-fraud): It has been shown that these fake identities often form a densely connected cluster which has only very few connections to the remaining network [4].

[2] `http://snap.stanford.edu/data/index.html`

For each of the scenarios we are interested in (1) the single thread overhead compared to the sequential (Seq) throughput, (2) the scalability that can be achieved by the underlying STM implementation, and (3) the contention on the state introduced by the parallel execution.

Our tests have been carried out on a dual-socket server with two 6-core Intel Xeon Westmere-EP X5650 running 64-bit Linux 3.5. All 6 cores of a processor share the L3 cache. The CPU affinity was configured such that the penalty of moving data between sockets is as limited as possible, i.e., for up to 6 threads only a single processor is used.

The DREAM benchmark was compiled with the DTMC open-source TM C/C++ compiler [6]. In combination with the framework that provides the thread-pool, queues and an operator interface, this dramatically simplified the development. The implementation of the operators feels like sequential code and contains transaction demarcations around the state access inside the operator. The parallelization is accomplish transparently for the programmer by dispatching the operator to all threads in the thread pool and the synchronization is handled by the STM.

3.1 Communities Of Interest

Figure 4 shows the throughput for the COI operator. The operator is read dominated and after an initialization phase that fills the top-k lists only the weights are updated and hardly any elements are replaced due to the power-law distribution of the data. This allows the majority of parameter settings to scale well with the number of threads. In the first two graphs (P=1 N=1024 W=10 K=9 and P=1 N=32768 W=20 K=9), the size of the communities is distributed equally among all possible target IDs. This prevents a fast saturation of the top-k list and results in larger write-sets.

FASTLANE scales well for all cores of a single socket but stagnates when threads are executed on two sockets because it relies on extensive usage of a shared counter. ETL and WT scale well for up to larger numbers of threads. NOREC outperforms

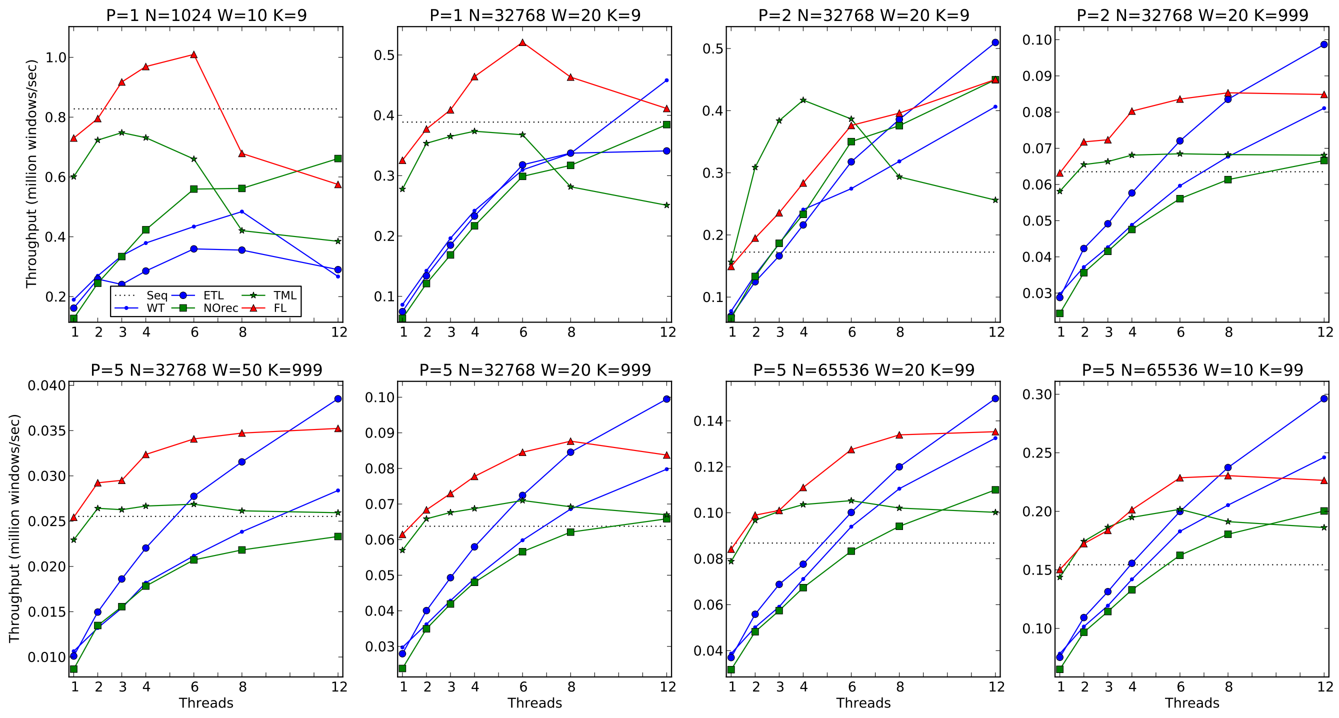

Figure 4. Throughput in merged windows per second for the COI operator.

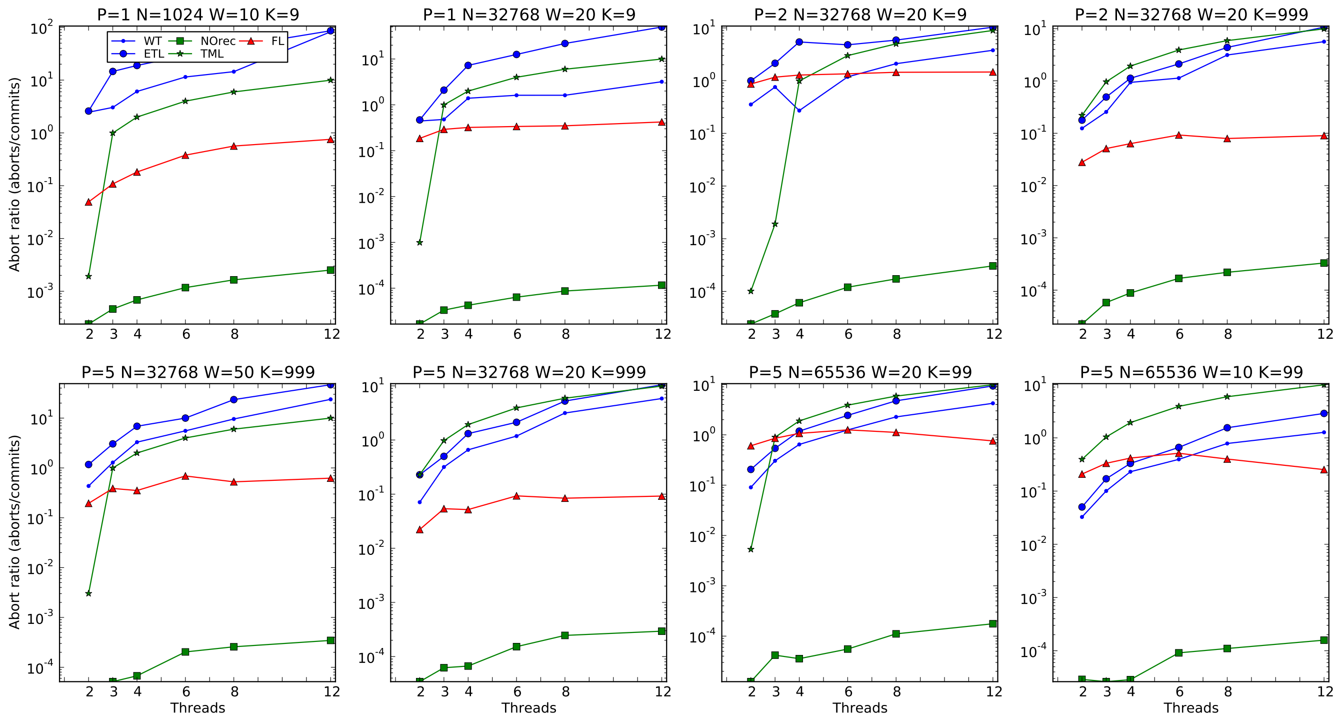

Figure 5. Abort ratio for the COI operator.

ETL and WT in the first graph because it benefits from value-based validation in the list updates. This is a hint that their internal locking scheme produces too many false conflicts. TML scales well when threads do not update memory concurrently but does not advance significantly beyond the sequential baseline when only a few concurrent updates are present.

Except for FASTLANE and TML, the STM implementations require in average one processor with 6 cores to beat the sequential throughput. This is due to the setup costs of the transaction and the high bookkeeping overhead for memory accesses. FASTLANE and TML reduce this overhead by omitting the bookkeeping of undo information for rollbacks and can outperform the sequential performance already with 2-3 threads.

Figure 5 shows the abort ratio for the COI operator. NOREC has a very low abort rate but spends a large fraction of time waiting for the global versioned lock. TML drastically increases in its abort rate already for a few threads because only one transaction can perform updates at a time and the others must abort. WT and ETL both show a similar trend on a high abort rate level. FASTLANE has a lower abort rate because one thread executes transactions pessimistically, i.e., they cannot abort.

3.2 Community Detection

Figure 6 shows the throughput for the LPA operator. The operator is write-dominated and hard to scale for smaller numbers of neighbor IDs (N). When more threads are added the contention increases and prevents further performance improvements. NOREC, ETL and WT even decrease in throughputs in some situations with high threads counts. Ideally, the throughput should stay on the level of the sequential baseline under high contention. FASTLANE achieves this for low thread counts but suffers when data is cached on both sockets because the transactions are relatively small. The operator provides a challenging synchronization workload.

Figure 7 shows the abort ratio for the LPA operator. All STM implementation suffer from a high abort rate because according to the power-law the threads try to append the community messages to only a small number of inboxes. FASTLANE has a decreasing number of aborts it implies per commit because it spends a large fraction of time waiting for data being transferred between the sockets.

While the overall scalability of the operator is limited, it provides a good stress test for the mapping from memory locations to meta data.

4. Related Work

In this section we first present the most commonly used benchmark applications that are employed to evaluate transactional memory algorithms. Thereafter, we continue with an overview over the state of the art in graph processing frameworks.

4.1 Transactional Memory Benchmarks

The synthetic *intset* micro-benchmarks perform randomly queries and updates on integer sets implemented as a *red-black tree*, a *linked list*, a *skip list*, or a *hash set*. Their workload can be specified with the update-to-lookup ratio and the number of elements in the set.

The realistic workloads from the STAMP [5] benchmark suite consist of the following applications: `bayes` learns the structure of Bayesian networks in a directed acyclic graph; `genome` performs gene sequencing using hash sets and string search; `intruder` emulates a signature-based network intrusion detection system by matching packets against signatures stored in self-balancing trees; `labyrinth` finds the shortest-distance paths between pairs of points using breadth-first search; `kmeans` clusters a set of partitioned points in parallel; `ssca2` constructs an efficient graph data structure using adjacency arrays; `vacation` emulates a travel reservation system, reading and writing different tables that are implemented as red-black trees; finally, `yada` performs mesh refinement of triangles in a work queue. The STAMP applications spend most time in transactions. The workload is partitioned by the number of threads at the start of each parallel region. This works because of the operations have an evenly distributed length.

The benchmark STMBench7 [15] adapted a workload that had originally the intention to compare object-oriented database systems. It is based on a rich object-graph and a large set of operations that access the graph. The operations can be classified into four categories: (1) long traversals access a large subset of the graph, (2) short traversals operate on random paths, (3) short operations access only a random element or neighborhood in the graph, and (4) modifying operations insert or remove elements or links in the graph. The underlying data structure is highly regular and the operations provide a high contention level. The distribution in the object-graph is balanced. The traversal of the operations on the graph is random but the properties of the graph are maintained during modifications.

The Lee-TM [2] benchmark suite is based on Lee's routing algorithm that computes interconnections between electronic components. It consists of an expansion phase that performs a breadth-first search and a backtracking phase that does the shortest path routing. It supports a wide range of transaction sizes and complex contention characteristics and allows to verify the computed result for correctness of the transactional memory implementation. The data distribution depends on the input circuit layout but is bound by the number of components that fit into the logic.

RMS-TM [20] is a benchmark suite consisting of seven real-world applications from the domain of data recognition, mining and synthesis: `hmmsearch` tries to match sequences; `hmmpfam` queries a database for a sequence; `hmmcalibrate` performs a profile calibration using shared counters; `apriori` is a data mining algorithm on data bases using a hash table; `scalparc` computes a decision tree by partitioning data into subsets; `utilitymine` finds high ranked items in a hash tree; and `fluidanimate` performs operations on the boundaries of data partitions. The majority of the applications does not spend all time in transactions and provides a very good scalability.

4.2 Graph Processing Frameworks

There are powerful distributed graph processing frameworks. The first one that had a major impact was Pregel [23]. The main challenge for distributed graph processing is, that most graph algorithms do very small computations on a large number of vertices. Moreover, it is often required to exchange information between vertices (such as label propagation). To this end, it was necessary to force a restricted programming paradigm onto the developers. In Pregel, general graph algorithms have to be rewritten into vertex programs - i.e. small functions which are scheduled by the framework on every vertex. Thus, the programmer has to care about how to effectively distribute an algorithm. This direction is further pursued by Powergraph [14] which is based on Pregel.

In contrast, shared memory, multi-threaded graph processing is in its infancy [22]. For example, the SNAP graph framework only supports a limited set of graph algorithms which have been extended for parallel use [3]. Finally, GraphChi [21] shows impressive performance but borrows the interface from Pregel and Powergraph and thus, requires the programmer to follow the same restricted programming paradigm.

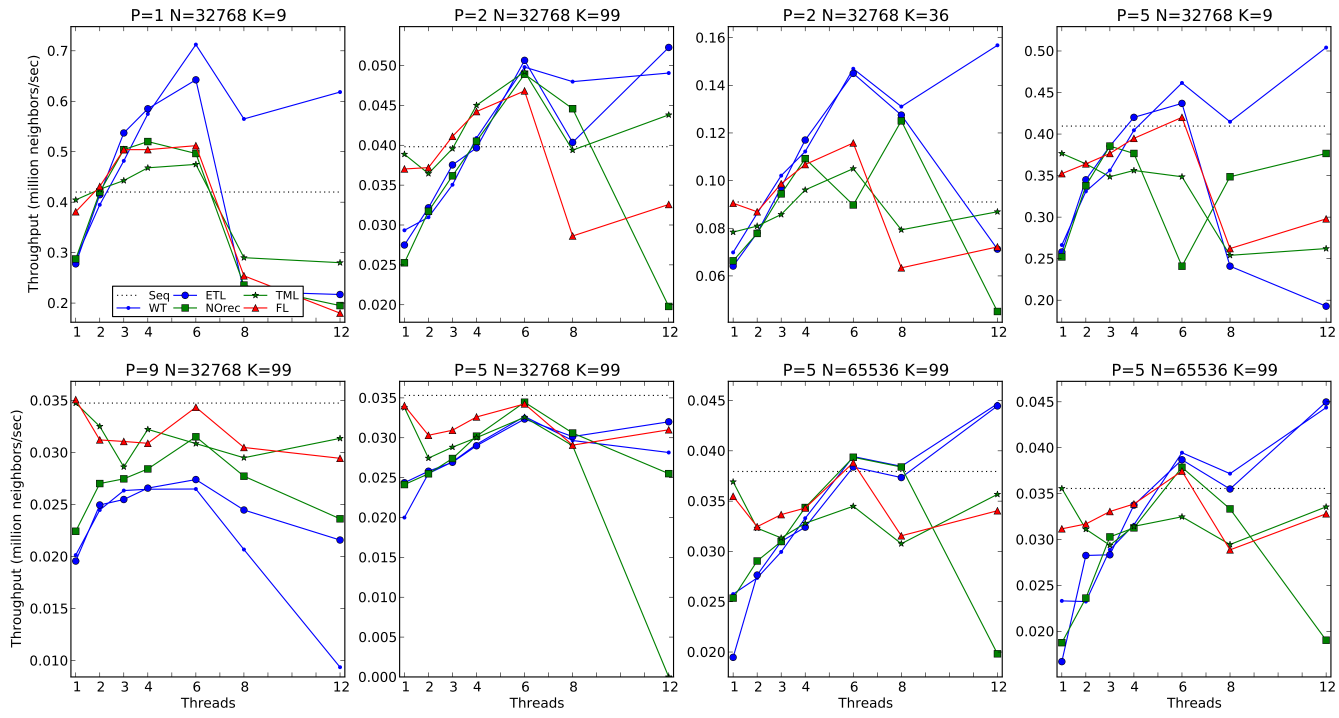

Figure 6. Throughput in appended neighborhoods per second for the LPA operator.

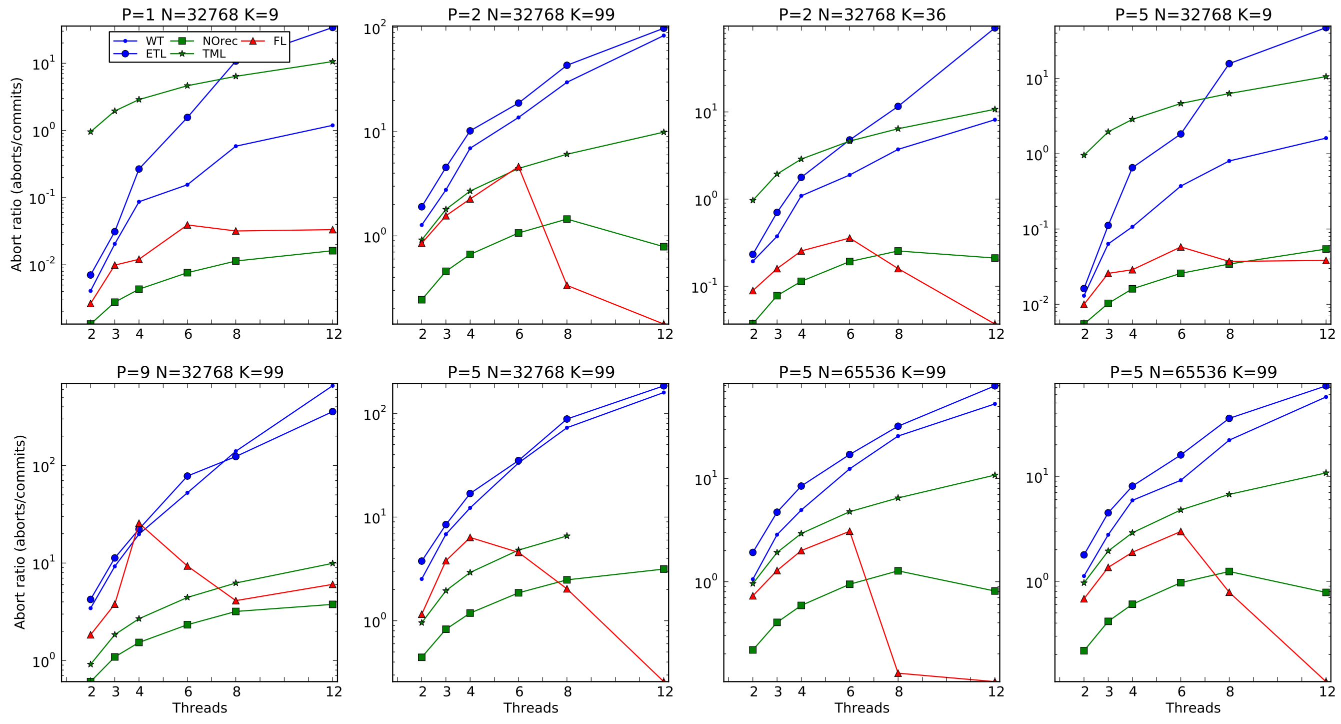

Figure 7. Abort ratio for the for the LPA operator.

5. Conclusion

We have presented the Dresden Streaming Transactional Memory Benchmark (DREAM)[3], a framework that allows a straight forward simulation of operators from the domain of data stream processing. We evaluated current software transactional memory implementations using two real-world workloads: a community of interest operator and a community detection operator based on label propagation. The benchmark is extensible to easily add new operators for the processing of stream updates to large graphs.

We are planning to add additional stream operators in future work that cover additional workload characteristics. The unbalanced workload is a good candidate for an in-depth study of the mapping of memory locations to ownership records in lock-based transactional memory implementations.

[3] `http://tm.inf.tu-dresden.de/`

References

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. CIDR, 2005.

[2] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis. Lee-tm: A non-trivial benchmark suite for transactional memory. In A. Bourgeois and S. Zheng, editors, *Algorithms and Architectures for Parallel Processing*, volume 5022 of *Lecture Notes in Computer Science*, pages 196–207. Springer Berlin Heidelberg, 2008.

[3] D. Bader and K. Madduri. Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –12, april 2008. doi: 10.1109/IPDPS.2008.4536261.

[4] Z. Cai and C. Jermaine. The latent community model for detecting Sybils in social networks. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, Feb. 2012.

[5] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[6] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 27–40, New York, NY, USA, 2010. ACM.

[7] C. Cortes, D. Pregibon, and C. Volinsky. Communities of interest. *Intelligent Data Analysis*, 6(3):211–219, 2002.

[8] L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear. Transactional mutex locks. In P. D'Ambra, M. Guarracino, and D. Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 2–13. Springer Berlin / Heidelberg, 2010.

[9] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: Streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[10] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 39–52, New York, NY, USA, 2011. ACM.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 194–208, September 2006.

[12] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, New York, NY, USA, 2008. ACM.

[13] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21:1793–1807, 2010.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, October 2012.

[15] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: A benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3): 315–324, 2007.

[16] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, December 2010.

[17] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[18] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. 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.

[19] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. Kirchner, and J. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 693–702. ACM, 2002.

[20] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. Rms-tm: A comprehensive benchmark suite for transactional memory systems. In *Proceedings of the second joint WOSP/SIPEW international conference on Performance engineering*, ICPE '11, pages 335–346, New York, NY, USA, 2011. ACM.

[21] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, October 2012.

[22] A. LUMSDAINE, D. GREGOR, B. HENDRICKSON, and J. BERRY. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007. doi: 10.1142/S0129626407002843. URL `http://www.worldscientific.com/doi/abs/10.1142/S0129626407002843`.

[23] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkõwski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807184. URL `http://doi.acm.org/10.1145/1807167.1807184`.

[24] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, pages 29–42, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-908-1. doi: 10.1145/1298306.1298311. URL `http://doi.acm.org/10.1145/1298306.1298311`.

[25] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, 2006.

[26] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, 76:036106, Sep 2007. doi: 10.1103/PhysRevE.76.036106. URL `http://link.aps.org/doi/10.1103/PhysRevE.76.036106`.

[27] T. Riegel, P. Felber, and C. Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, pages 284–298, 2006.

[28] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *SPAA '11: Proceedings of the twenty-third annual symposium on Parallelism in algorithms and architectures*, New York, NY, USA, 2011. ACM.

[29] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, Aug 1995.

[30] J. Shun and G. E. Blelloch. Fastlane: Improving performance of software transactional memory for low thread counts. In *PPoPP '13: Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, Shenzhen, China, 2013. ACM.

[31] J.-T. Wamhoff, C. Fetzer, P. Felber, E. Rivière, and G. Muller. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP '13: Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, Shenzhen, China, 2013. ACM.

[32] S. Weigert, M. Hiltunen, and C. Fetzer. Mining large distributed log-data in near real-time. In *Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques (SLAML/SOSP)*, SLAML '11, pages 5:1–5:8, New York, NY, USA, October 2011. ACM. ISBN 978-1-4503-0978-3. doi: 10.1145/2038633.2038638.