FASTLANE: Improving Performance of Software Transactional Memory for Low Thread Counts

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

Pascal Felber
Etienne Rivière
Université de Neuchâtel, Switzerland
first.last@unine.ch

Gilles Muller
INRIA, France
first.last@inria.fr

Abstract
Software transactional memory (STM) can lead to scalable implementations of concurrent programs, as the relative performance of an application increases with the number of threads that support it. However, the absolute performance is typically impaired by the overheads of transaction management and instrumented accesses to shared memory. This often leads STM-based programs with low thread counts to perform worse than a sequential, non-instrumented version of the same application.

In this paper, we propose FASTLANE, a new STM algorithm that bridges the performance gap between sequential execution and classical STM algorithms when running on few cores. FASTLANE seeks to reduce instrumentation costs and thus performance degradation in its target operation range. We introduce a novel algorithm that differentiates between two types of threads: One thread (the master) executes transactions pessimistically without ever aborting, thus with minimal instrumentation and management costs, while other threads (the helpers) can commit speculative transactions only when they do not conflict with the master. Helpers thus contribute to the application progress without impairing on the performance of the master.

We implement FASTLANE as an extension of a state-of-the-art STM runtime system and compiler. Multiple code paths are produced for execution on a single, few, and many cores. The runtime system selects the code path providing the best throughput, depending on the number of cores available on the target machine. Evaluation results indicate that our approach provides promising performance at low thread counts: FASTLANE almost systematically wins over a classical STM in the 1-6 threads range, and often performs better than sequential execution of the non-instrumented version of the same application starting with 2 threads.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming

General Terms Algorithms, Performance.

Keywords Transactional Memory, Concurrency.
they can. The latter threads, called helpers, typically run slower than STM threads, because in addition to performing the extra bookkeeping associated with memory accesses they should not hamper progress of the master. The roles of master and helper can be changed dynamically by the runtime system during execution of the concurrent application, e.g., if a thread requests to perform irrevocable operations and must execute as master.

An application compiled for FASTLANE includes the different synchronization strategies (sequential, STM, and FASTLANE) to allow the selection of the appropriate strategy depending on the number of cores available on the target machine. For this, we have extended the DTMC compiler [2] so as to generate all the synchronization strategies within the application binary.

We have evaluated the performance of FASTLANE on a number of synthetic and realistic benchmarks, and compared them against STM and sequential executions. Our results show that FASTLANE performs competitively with sequential execution for a single thread, and performs most of the time better with already two threads. Further, FASTLANE often continues to scale well and generally outperforms STM algorithms up to six threads, which corresponds to the number of cores per processor on our test machine. When the workload can be partitioned, FASTLANE can execute one master thread per partition and performs significantly better than other STM algorithms even for high thread counts.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the FASTLANE algorithm and the design choices that led to several optimizations. Section 4 evaluates the performance of the algorithm on various synthetic and realistic benchmarks. Finally, Section 5 concludes.

2. Related Work

A wide variety of efficient software transactional memory implementations have been proposed over the last few years [5, 6, 8, 9, 11, 25]. The main focus has been on exploiting the available disjoint access parallelism with high thread counts. It has also been shown in previous work that dynamic tuning of the STM runtime system depending on the workload can significantly improve the throughput [20, 22, 26], e.g., the bookkeeping overhead can be reduced when no contention is present. Instead of tuning, FASTLANE focuses on optimizing the synchronization algorithm for few threads.

We are aware of only few STM designs that explicitly target small thread counts. Transactional mutex locks (TML) [4] use a versioned reader-writer lock: read-only transactions can concurrently execute in parallel. TML can execute every transaction once and never aborts, enabling the execution of irrevocable operations and simpler debugging at the cost of limiting concurrent updates. ROBUST [27] starts to execute transactions speculatively but gives transactions that aborted a certain number of times a priority privilege. This privilege lets a transaction pessimistically win all conflicts, even in the presence of crashes and non-terminating transactions, making it practically wait-free. Similarly, the authors of [13] propose to execute a transaction subject to a deadline in increasingly pessimistic modes as that deadline nears.

Runtime systems for parallelization often use speculation. Software lock elision [21] processes critical sections speculatively in parallel but will fall back to lock acquisition upon frequent conflicts or irrevocable operations, which always wins over speculation. Other runtime systems auto-parallelize programs by thread-level speculation [18] or profile-guided automatic loop parallelization [15]. Fastpath [24] uses pessimistic and speculative modes to parallelize loops. Each iteration starts in speculative mode, using NOREC for synchronization, but can switch to a pessimistic mode that requires only entry and exit instrumentation whenever proceeding iterations have finished.

3. FASTLANE Algorithms

The high-level objective of FASTLANE is to perform (1) approximately identically to sequential execution, and (2) better when leveraging a few additional threads. To meet the first goal, we rely on a pessimistic and lightly-instrumented master thread that never aborts and, hence, should provide performance similar to sequential execution on a single core. The role of the helper threads is to address the second objective, i.e., improve performance by committing transactions that do not conflict with those of the master.

We start by describing the global data structures used by FASTLANE and the behavior of the master thread, before describing the helper threads and the optimizations we applied.

3.1 Data Structures

The shared data structures used by the FASTLANE algorithms are summarized in Table 1. They essentially consist of: a shared counter, counter, that is incremented each time a transaction commits updates; a shared array of integers, dirty, that protects a set of memory addresses and stores the value of the counter at the last
Lock to serialize commit attempts of helpers.

*counter that tracks updates of the master and helpers.
The value is odd when a transaction performs updates and even otherwise. This variable is used for validation by the helpers.

**dirty[]**
Array of monotonically increasing integers. Each memory address is mapped to one entry in the array (by hashing the address modulo the size of the array). The entry contains the value of the counter counter at the last time the address was written.

**helpers**
Lock to serialize commit attempts of helpers. It is implemented as a MCS list-based queue lock [16] and provides FIFO guarantees.

**master**
Lock to synchronize the master with the helper. Helpers must acquire helpers first. It is implemented as a TTAS (test-and-test-and-set) lock [12] and protects all shared variables.

**masterID**
Identity of the current master thread. It must only be modified after the master has been acquired.

Table 1. Shared variables used by FASTLANE algorithms.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter</td>
<td>Counter that tracks updates of the master and helpers. The value is odd when a transaction performs updates and even otherwise. This variable is used for validation by the helpers.</td>
</tr>
<tr>
<td>dirty[]</td>
<td>Array of monotonically increasing integers. Each memory address is mapped to one entry in the array (by hashing the address modulo the size of the array). The entry contains the value of the counter counter at the last time the address was written.</td>
</tr>
<tr>
<td>helpers</td>
<td>Lock to serialize commit attempts of helpers. It is implemented as a MCS list-based queue lock [16] and provides FIFO guarantees.</td>
</tr>
<tr>
<td>master</td>
<td>Lock to synchronize the master with the helper. Helpers must acquire helpers first. It is implemented as a TTAS (test-and-test-and-set) lock [12] and protects all shared variables.</td>
</tr>
<tr>
<td>masterID</td>
<td>Identity of the current master thread. It must only be modified after the master has been acquired.</td>
</tr>
</tbody>
</table>

The operation of the master thread is described in Algorithm 2. After the selection of the master code path in Section 3.2 the thread already owns master at MASTERSSTART (line 2) and has exclusive privilege to directly update shared data during the pessimistic transaction.

Instrumentation of memory accesses is minimal on the master thread so as to obtain performance as close as to a single thread.

3.2 Code Path Selection
We have extended the DTMC open-source C/C+++ TM compiler [2] to generate multiple code paths for each transaction (see Figure 2): (1) a sequential path without instrumentation of reads and writes, (2) a pessimistic master path with a lightweight instrumentation of writes, (3) a speculative helper path with instrumentation for reads and deferred writes, and (4) an optimistic fully-fledged STM path with instrumentation of reads and writes.

The selection of the code path is performed dynamically at the beginning of a transaction. As long as only a single thread is registered with the runtime system, the sequential code path will be executed. The FASTLANE mode with the master and helper code paths is enabled after a second thread is active. When more than a specified number of threads are registered, the STM mode is selected. This number is currently fixed empirically based on observations of the average number of threads after which STM outperforms FASTLANE. Switching to STM mode requires the acquisition of a quiescence lock that prevents further helper transactions from starting, as well as the acquisition of the master lock to stop the master. When no threads execute transactions, it is safe to set the code path for all subsequent transactions to the STM algorithm.

Algorithm 1 shows the selection of the master and helper in the FASTLANE mode. Initially, the thread that registered first with the runtime system becomes the master. When the master calls START to begin a transaction (lines 1–2) it must first acquire master using TTAS to synchronize with the helpers (line 3). As the master transaction will update applies in-place during its execution,

3 An MCS lock uses a list-based queue to grant access to the lock in FIFO order. Threads only spin on cacheable local memory, thus achieving high scalability.

4 A TTAS (test-and-test-and-set) lock scales better than a simpler test-and-set spin lock. TTAS first reads the state of the lock using a normal memory access and, if it is free, tries to acquire it using an atomic operation. This helps avoid unnecessary traffic on the bus because the requesting thread will spin on its local cache if the lock is not free. A TTAS lock can be released by a simple write to memory.

Figure 2. Multiple code paths are generated for each transaction and can be selected at transaction start by the runtime system.

master must be owned during that time. If another thread was meanwhile promoted to master, the thread releases master and continues as helper (lines 7–10, Section 3.4). For the common case that the master remains unchanged, one can assume that masterID is cached, hence, the overhead of the additional check is negligible. The thread can continue as master (lines 4–6, Section 3.3). Note that transactions that have an explicit abort request (cancel) cannot be executed as master. The compiler identifies such transactions and generates code that always selects the helper code path.

Helper threads will acquire master during their HELPER-COMMIT because they execute speculatively in parallel with the pessimistic master. They directly jump to the helper code path (lines 15–17, Section 3.4).

Helper threads can also request to gain master privilege, e.g., to perform irrevocable operations such as I/O or system calls (see Section 3.5). Upon such an event, the thread aborts and sets the pessimistic flag when the transaction is restarted (line 11). A thread that wants to become master first acquires master using TTAS (line 12). It can now execute in isolation and set masterID to its thread identifier to reflect the change (line 13). It then continues with the master code path (line 14).

Algorithm 1: Code path selection at transaction start.

```
1 function START(pessimistic) // Begin or restart a transaction
2 if masterID = threadID then // Is thread currently master?
3 ttas-lock(master) // Acquire master
4 else if masterID = threadID then // Is thread still master?
5     masterID ← threadID // Set current thread as master
6     return CP_MASTER // Jump to master code path
7 else ttas-unlock(master) // Release master
8     HELPERSTART // Start transaction as helper
9     return CP HELPER // Jump to helper code path
10 if pessimistic then // Master privilege requested?
11     ttas-lock(master) // Acquire master
12     masterID ← threadID // Set current thread as master
13     return CP_MASTER // Jump to master code path
14 else // Thread is helper
15     HELPERSTART // Start transaction as helper
16     return CP HELPER // Jump to helper code path
```
case. `MASTERREAD` operations are not instrumented (line 4) because the master does not need to ever validate the read set, while `MASTERWRITE` operations are augmented of a store of the value of the counter in the corresponding entry of the `dirty` array (line 8). No undo logging is required because the pessimistic master never aborts. On the first write only, `cntr` must be incremented to an odd value (line 7). Finally, upon `MASTERCOMMIT`, the master simply reverts the counter to an even value in case it has performed writes (line 12) and releases `master` (line 13).

In most cases, the master has very low overhead. The `masterID` variable is infrequently modified and thus remains in the CPU cache. At transaction `START` only one atomic `test-and-set` operation is needed to implement the TTAS lock. Between the TTAS lock and unlock operations, only the master can write shared data and thus we do not need any additional atomic operations or barriers. `MASTERREAD` operations are not instrumented and `MASTERWRITE` operations go directly to memory. Upon the first write `cntr` is incremented using a simple store, which may cause invalidation messages if `cntr` is cached in other cores. Note that `cntr` cannot be updated by other threads while `master` is owned, thus subsequent `MASTERWRITE` operations have lower overhead because `cntr` is cached and unmodified. If the transaction is read-only we completely avoid the invalidation of `cntr`. Finally, one update to the `dirty` array is necessary for every write. Upon `MASTERCOMMIT`, the `master` lock is released in order to serialize all changes made by the transaction. In Section 3.6 we show how the contention on `master` and `cntr` can be reduced.

### 3.4 Helper Thread

The price to pay for having a lightly instrumented master thread becomes clear when considering the algorithm of the helpers. Extra work must be performed to speculatively execute transactions and try to commit changes without slowing down the master.

The functions of the FASTLANE helper code path are shown in Algorithm 3. Upon `HELPERSTART`, the current value of `cntr` is stored for subsequent validation purposes, discarding the least significant bit to force the value to be even (line 3).

For `HELPERREAD` operations, the helper first checks whether it has already written to the same address. If so, it returns the value of the previous write (lines 5–6). Otherwise, it reads the value and conservatively checks if the address has been concurrently written, by validating the associated entry of `dirty`: if so, the transaction simply aborts (lines 7–9). This guarantees opacity [10]. Otherwise, the read can successfully complete: the address is added to the read set and the previously read value is returned (lines 10–11).

Upon `HELPERWRITE`, we check if the written address has possibly been updated concurrently, like for reads, and if so, the transaction aborts (lines 13–14). Otherwise, we simply add or update the address and the written value in the write set (line 15), delaying the actual update of the shared memory to the commit phase.

The `HELPERREAD` and `HELPERWRITE` operations must both perform lookups on the write set for each invocation. The write set is a vector with an index to reduce the lookup time, following the same general principle as in [23].

The main idea of `HELPERCOMMIT` is to perform the validation of the read and write sets, resulting in either an abort or a successful commit, while holding `master`. If the transaction is read-only (lines 24–25), all memory accesses have already been validated by the `HELPERREAD` operation and the transaction can commit immediately. Otherwise, the helper must ensure mutual exclusion for its commit phase. To that end, it first acquires the MCS queue lock helpers (line 26) to synchronize with other helpers and then acquires `master` using TTAS (line 27) to synchronize with the master. The rationale behind using the additional `helpers` lock is to reduce the contention on `master`, i.e., minimize the negative impact of the helpers on the master. In Section 3.6 we use the `helpers` for a lock handover optimization.

The validation is performed while holding `master` and no other thread (even the master) can interfere. `VALIDATE` verifies if any address stored in the read and write set may have been concurrently
updated, by looking into the \texttt{dirty[]} array (lines 16–22), and if so \texttt{HELPERCOMMIT} conservatively aborts after releasing \texttt{helpers} and \texttt{master} (lines 28–31). Upon successful validation, all pending updates stored in the write set are sent to shared memory (line 35) and the associated entries of the \texttt{dirty[]} array are updated with the current odd \texttt{cntr} (lines 32 and 34). Finally, \texttt{cntr} is increased to the next even value (line 36) and \texttt{master} and \texttt{helpers} are released (lines 37–38).

### 3.5 Irrevocability And Privatization

Some operations cannot be executed speculatively because they cannot be reverted, e.g., system calls or other operations with externally visible side effects. If a thread encounters such an operation and is currently executing speculatively, it requests to enter pessimistic mode to ensure that the transaction must not abort. Depending on the current code path, this requires for a helper thread to become the master and for an STM thread to execute exclusively in sequential mode. Note that switching to the pessimistic modes can also be used when the progress of a thread is at stake, with the benefit of the master to allow parallel helpers in contrast to the exclusive sequential mode.

A thread in STM mode that requests irrevocability must acquire a quiescence lock, a common approach described in detail in the literature [28]. In short, it will prevent master threads from starting transactions. It then waits until all active transactions are either committed or aborted. After that it can execute the transaction in isolation, using a non-instrumented code path, and release the quiescence lock after commit.

In FASTLANE mode, the pessimistic master is used to execute irrevocable operations that cannot be rolled back. Helpers must abort the transaction when they encounter irrevocable operations, switch to the master code path (presented in Section 3.2) to re-execute the transaction. Other helpers can continue to execute transactions speculatively in parallel. Note that the operations of the master must not contain non-transactional code that does not reflect updates to shared memory in \texttt{dirty[]}.

Threads can request to privatize data in order to access it outside of transactions afterwards. Privatization safety [11] requires that no other threads are accessing the data after the privatizing transaction committed. This is supported natively for the master because, once it has started its privatizing transaction, no helper can commit concurrently before it is finished. The helpers will abort when their validation encounters data privatized by the master. Helper threads that request privatization must wait after they committed until all earlier helper transactions are either committed or aborted. This is determined by looping through all transaction descriptors and waiting if they are active and still have a start counter value less than the privatizing transaction.

### 3.6 Optimizations

The goal of FASTLANE is to reduce the overhead for master transactions in order to achieve a performance close to sequential non-instrumented execution. The lightweight instrumentation of the master presented in Section 3.3 combined with a distinct compiler generated code path is key to a range of optimizations: all master-specific instrumentation can be inlined to remove call overheads; because the master only updates global data, it does not need access to a transaction descriptor during its read or write operations; and at transaction start, the master can omit saving the context because it never aborts.

Helper threads that execute in parallel will have an impact on the master because \texttt{master}, \texttt{cntr} and \texttt{dirty[]} are shared resources. While the contention on \texttt{dirty[]} is spread over the elements of the array, the contention on \texttt{master} and \texttt{cntr} can have a negative impact because they will be modified by each update transaction.

With high likelihood, \texttt{cntr} will be in the cache of the helper threads for validation purposes, thus the modifications will cause invalidation traffic.

We applied a number of specific optimizations to the FASTLANE algorithm to further reduce the overheads of the master and helper threads:

1. **Keep-lock** — The master thread can keep the \texttt{master} lock if no helper requests it to avoid unnecessary updates to shared variables.
2. **Pre-validate** — Helper threads can validate before attempting to acquire the \texttt{master} lock, to stop the master only if they have a high likelihood of a successful commit.
3. **Hand-over** — Helper threads can hand over the \texttt{master} lock if a successor exists in the \texttt{helpers} MCS queue to increase the commit chance.

The first optimization (\textit{keep-lock}) can save the master the cost of releasing and reacquiring \texttt{master}, along with incrementing \texttt{cntr}, as long as no helper requests it. After each \texttt{MASTERCOMMIT}, the master checks if \texttt{helpers} was acquired.\footnote{It is sufficient to check if the head node of the \texttt{helpers} MCS queue is not null.} If this is not the case, the master keeps \texttt{master} and does not increment \texttt{cntr} to the next even value (line 3 in Algorithm 1 and lines 11–13 in Algorithm 2 will not be executed). A helper acquires \texttt{helpers} when it needs to commit and must acquire \texttt{cntr}, or when the helper aborts because it cannot validate. In the latter case, the helper needs the counter to be incremented to eventually commit. Indeed, assume that a shared variable has been written by the master when the value of the counter is \(x\) (odd) without the counter being subsequently incremented. A helper that later reads the same variable will remember \(x−1\) (even) as start value of the counter and will systematically fail validation until \texttt{cntr} becomes greater than \(x\) (see Algorithm 3, lines 3 and 20).

Helper threads can reduce the contention on \texttt{cntr} by validating before they attempt to acquire \texttt{master} with the second optimization (\textit{pre-validation}). After acquiring \texttt{helpers} (Algorithm 3, line 26) the thread calls \texttt{VALIDATE} and keeps the current value of \texttt{cntr}. Upon successful validation it acquires \texttt{master} (line 27) and must only validate again (line 28) if \texttt{cntr} was incremented because another thread committed in the meantime. This is expected to reduce contention with the master, as a transaction that is known to abort will not compete for \texttt{master}. We prevent multiple helpers from committing concurrently using the \texttt{helpers} lock in order to avoid interference with the \textit{pre-validation} from other helpers.

Finally, the third optimization (\textit{hand-over}) allows the helper threads to hand over the \texttt{master} lock if a successor is waiting in the \texttt{helpers} MCS queue at \texttt{HELPERCOMMIT}. In that case, the helper does not increment \texttt{cntr} to the next even value and skips releasing the \texttt{master} lock (Algorithm 3, lines 36–37). It only re-leases \texttt{helpers} (line 38) and the succeeding owner in the queue of \texttt{helpers} can skip the acquisition of \texttt{master} (line 27) and will increment \texttt{cntr} by two to the next odd value (instead of line 32). Besides a reduction of the contention on \texttt{master}, this optimization is expected to improve the chance for helper threads that have already reached the commit phase to complete it. This is particularly interesting when combined with the second optimization: \textit{pre-validation} holds because no other thread can commit in the meantime.
4. Evaluation

In this section, we evaluate and analyze the performance of FASTLANE. We compare the FASTLANE algorithm against non-instrumented sequential execution; two STM variants that are based on the lazy snapshot algorithm [19]: TINYSTM [8] operating either in write-through mode (WT), i.e., direct updates to memory, or in write-back mode with encounter time locking (ETL), i.e., buffered updates with eager conflict detection; and two STMs based on a single versioned lock, either exclusively directly updating memory (TML [4]) or performing buffered updates and value-based validation (NOREC [5]). For FASTLANE, we measured the plain algorithm (FL, see Sections 3.3 and 3.4) as well as the optimizations described in Section 3.6. All evaluated configurations are summarized in Table 2.

For our test applications, we use the synthetic intset micro-benchmarks, realistic applications from the STAMP [17] benchmark suite, and a new benchmark that we designed to analyse the benefits and limitations of FASTLANE and which computes communities of interest for communication networks (see Section 4.5).

The intset benchmarks perform randomly queries and updates on integer sets implemented as a red-black tree (RB), a linked list (LL), a skip list (SL), or a hash set (HS). We use a working set of 8,192 elements for RB; 2,048 and 1,024 elements for LL; and 1,024 elements for SL and HS. We use update-to-lookup ratios of 5% and 20%, and the execution time for each run is 10 seconds.

The STAMP benchmark suite consists 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 among pairs of points with breadth-first search; mmeans 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.

Using FASTLANE, there is a very unbalanced workload distribution between threads because the master is able to process transactions much faster than the helpers. Therefore we have adapted the STAMP benchmarks with a partitioning-based dynamic work balancing that introduces only very little overhead and allows adapting the amount of work for each thread and account for differences in throughput between the master and the helpers. Otherwise, all STAMP benchmarks are configured accordingly to the documentation with parameters for non-simulator runs and high contention.

Our tests have been carried out on a dual-socket server with two 6-core Intel Xeon Westmere-EP X5650 running 64-bit Linux 3.0. 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. All benchmarks were compiled with the DTMC open-source TM C/C++ compiler [2].

In the rest of this section, we first analyze the scalability of FASTLANE against existing STM algorithms for a low number of threads. Then we study the comparative contribution of the master and the helper threads. We later study the impact of optimizations. Finally, we present the new benchmark that computes communities of interest for communication networks.

### 4.1 Scalability for Low Thread Counts

Our main goal is to achieve better scalability for a low thread counts than traditional STM approaches. Figure 3 presents the throughput obtained in millions of transactions per second for the duration of each of the intset benchmarks. Figure 4 presents completion times for STAMP applications, which have a fixed number of transactions depending on the input parameters. The STAMP graphs show execution times instead of scalability to allow an easy comparison with the sequential baseline Overall, FASTLANE scales well for a number of threads up to 6 and is on average more efficient than STM approaches for that number of threads. The sequential baseline corresponds to an implementation of transactions using a global lock because all benchmarks spend most of their time inside transactions and no scalability can be achieved.

On the intset micro-benchmarks, the minimal overhead of the master thread gives it a head start and the helpers contribute their share when the number of threads increases. FASTLANE is only outperformed on the RB micro-benchmark by TML, because of the instrumentation overhead to access dirty[1] in FASTLANE. TML is typically very efficient for workloads that contain mostly transactions that are either short, read-only, or have a short period of updates at the end, but cannot exploit parallelism as long as a single update transaction is active.

The intset micro-benchmarks have a drop in throughput when more than 6 threads are active and the second socket is in use. This due to the more expensive cache coherence traffic over the interconnect, for up to 6 threads the L3 cache can handle the coherence. Here, the main source of cache contention, common to all STMs and FASTLANE, is the shared clock used for versioning that must be incremented upon each update transaction.

The STAMP benchmarks do not exhibit this behavior because they have larger transactions and, hence, the relative impact of the cache contention bottleneck is lower. Here, FASTLANE wins over all STMs for up to 6 threads with the exception of labyrinth, intruder and ssca2 for which ETL is more efficient. With the two latter benchmarks, the master performs short update transactions at a high rate that prevent the helpers from committing.

Note that the performance of some algorithms is not shown for bayes and labyrinth because the DTMC compiler instruments loading of regions, which is currently not supported by NOREC and results in prohibitively long execution times for WT. Hence, both variants are omitted from the graphs. The other algorithms serialize transactions upon such an operation: FASTLANE switches to master mode; ETL acquires the quiescence lock and executes the non-instrumented sequential code path; and TML acquires its reader-writer lock. As a result, labyrinth does not exhibit scalability and the plot only shows the constant instrumentation

### Table 2. STMs and FASTLANE configurations used in the tests.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>Non-instrumented sequential execution.</td>
</tr>
<tr>
<td>WT</td>
<td>TINYSTM operating in write-through mode (direct updates to memory) [8].</td>
</tr>
<tr>
<td>ETL</td>
<td>TINYSTM operating in write-back mode with encounter time locking (buffered updates with eager conflict detection) [8].</td>
</tr>
<tr>
<td>TML</td>
<td>Single versioned lock with exclusive direct updates to memory [4].</td>
</tr>
<tr>
<td>NOREC</td>
<td>Single versioned lock with buffered updates and value-based validation [5].</td>
</tr>
<tr>
<td>FL</td>
<td>Basic version of FASTLANE with no optimizations (Sections 3.3 and 3.4).</td>
</tr>
<tr>
<td>FL-PV</td>
<td>FASTLANE with pre-validation before the helpers stop the master (Section 3.6).</td>
</tr>
<tr>
<td>FL-HO</td>
<td>FASTLANE with hand-over of locks for helper threads and keep-lock for master threads (Section 3.6).</td>
</tr>
<tr>
<td>FL-OJ</td>
<td>FASTLANE with all three optimizations enabled: keep-lock, pre-validation, and hand-over (Section 3.6).</td>
</tr>
<tr>
<td>FL-P</td>
<td>FASTLANE with support for partitions (one master per partition, see Section 4.5).</td>
</tr>
</tbody>
</table>
overhead for the FASTLAME master and TML, while ETL executes non-instrumented code. Bayes scales beyond sequential execution because not all time is spent inside serialized transactions, leaving a portion of the application where parallelism can be exploited.

Even if FASTLANE shows better performance than STMs on yada, it does not scale. The reason is that yada spends most of its time in long-running transactions and that FASTLANE serializes transactions of the master thread and commits of helper transactions by the master lock. As long as the master executes a transaction, no helper can commit and when a helper wants to commit it must stop the master.

4.2 Contribution of the Master

To better understand the performance of FASTLANE, we first evaluate the overhead of the master thread with respect to the non-instrumented sequential execution. Since we run the plain FASTLANE algorithm without dynamic switching of code paths, using one thread amounts to using only the master thread. In that case, instrumentation is lightweight: it only needs to acquire and release master upon beginning and committing a transaction, respectively. Loads have no instrumentation at all, while writes only require an additional update to the dirty[] array and the first write additionally increments cntr.

The one thread results in Figure 3 and Figure 4 show that the master can indeed achieve single-threaded throughput close to that of sequential execution. For inset, the performance is very close to sequential: less than 2% slower for LL, at most 34% for RB, and 16% on average. For STAMP, the overhead ranges from 5% for Genomes to 52% for yada, with an average of 29%.

This good performance can be explained because master and cntr are cached and have only a marginal impact on the overhead. With an increasing update rate the overhead slightly increases because dirty[] must be updated more often. Since the optimization that keeps master does not perform noticeably faster than the basic algorithm, the updates to dirty[] are the main source of overhead for the master.

When comparing FASTLANE to the state-of-the-art STM algorithms, the latter require non-trivial algorithms to be executed for every transactional operation. While runtime systems could decide to choose the sequential non-instrumented path if only a
single thread is active, we are interested in the general overhead introduced by an algorithm during its normal operation, i.e., the instrumented code path but without contention. STMs must typically copy the current CPU context at transaction START to support restart upon abort, keep track of read and write sets upon memory accesses, and perform validation and memory copy operations upon commit. FASTLACE’s objective is to streamline these costs for the master thread.

TinySTM and NOREC suffer from high transactional management costs that are mainly depending on the number of transactional memory accesses, e.g., LL has the largest transaction sizes and the biggest performance degradation. TML has slightly higher overhead than FASTLACE because it must additionally save the context at START and instrument reads to check for concurrent writers. These observations are also visible in the STAMP measurements. Only bayes and labyrinth differ, because they contain compiler instrumented regions that will be executed in irrecoverable mode by ETI (non-instrumented) and by the master with constant overhead in FASTLACE.

4.3 Contribution of the Helpers

To understand the importance of helpers in the global commit throughput of applications, we show in Table 3 the percentage of commits that have been achieved by helpers for all FASTLACE variants and all applications, with 2 and 6 threads. For the intset micro-benchmarks the contribution of the helpers varies from approximately 15-40% with 2 threads, to 60-85% with 6 threads. There is no noticeable difference between the FASTLACE variants. This can be explained by the fact that transactions are short and all identical, which limits the benefits of pre-validation and lock hand-over.

For the STAMP benchmark, one can observe important differences between the applications and the FASTLACE variants. Helpers contribute almost nothing with the bayes, intruder, and yada applications because of the size of transactions that leave almost no opportunity for helpers to commit without conflicts. One should also point out that bayes and labyrinth have very few transactions (less than 2,000) for the whole execution) and, hence, results are not very representative.

With genome, ssca2, and vacation, we observe the benefits of the hand-over optimization. When executing with 6 threads, the contribution of helpers in genome increases from 6.3% to 50.9%, i.e., almost one order of magnitude. Gains are also significant with the other two applications.

Table 3. Contribution of helpers to the global commit throughput.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Helper commits (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 thr</td>
</tr>
<tr>
<td>rb 8k 0%</td>
<td></td>
</tr>
<tr>
<td>rb 8k 20%</td>
<td></td>
</tr>
<tr>
<td>ll 1k 0%</td>
<td></td>
</tr>
<tr>
<td>ll 2k 0%</td>
<td></td>
</tr>
<tr>
<td>sl 1k 0%</td>
<td></td>
</tr>
<tr>
<td>sl 1k 20%</td>
<td></td>
</tr>
<tr>
<td>ha 1k 0%</td>
<td></td>
</tr>
<tr>
<td>ha 1k 20%</td>
<td></td>
</tr>
</tbody>
</table>

| bayes     | 1.7  | 2.3  | 1.7  | 5.0  | 1.1  | 2.5  | 1.1  | 7.3  |
| genome    | 3.8  | 6.3  | 3.5  | 6.3  | 18.4 | 50.9 | 24.1 | 53.2 |
| intruder  | 1.3  | 1.6  | 1.4  | 2.2  | 0.9  | 1.5  | 0.0  | 3.6  |
| labyrinth | 22.9 | 51.8 | 23.6 | 47.3 | 19.8 | 63.6 | 21.5 | 62.1 |
| kmeans    | 12.2 | 1.4  | 14.5 | 29.0 | 0.3  | 1.5  | 0.3  | 22.1 |
| ssca2     | 12.5 | 44.8 | 21.9 | 61.0 | 34.4 | 58.3 | 34.4 | 58.3 |
| vacation  | 28.1 | 50.0 | 25.0 | 49.0 | 31.2 | 69.8 | 31.2 | 72.9 |
| yada      | 0.2  | 0.0  | 0.2  | 0.0  | 0.0  | 0.1  | 0.1  | 0.5  |

Table 4. Impact of optimizations for FL-O3: percentage of failed pre-validations; percentage of pre-validations that hold after acquisition of the master lock; and hand-overs of helpers.

Interestingly, kmeans appears to suffer from the hand-over optimization but benefits from pre-validation. Indeed, with 6 threads, the contribution of the helpers increases from 1.4% to 29% when activating the latter optimization. This is because pre-validation helps detect early that a transaction is doomed and must abort, without needing to acquire the locks and slow down other threads. This optimization is particularly important given the low level of contention in kmeans.

4.4 Impact of Optimizations

We now focus on the the impact of optimizations. Table 4 shows, for FL-O3, the following metrics for 2, 6, and 12 threads: (1) percentage of failed pre-validations, i.e., helper transactions abort before even trying to acquire the master lock and hence do not slow down the master; (2) percentage of pre-validations that still hold after acquisition of the master lock, i.e., helper transactions do not need to validate again; and (3) percentage of committed helper transactions that benefited from the hand-over optimization.

One can first notice that pre-validation almost never fails for the intset benchmarks and several of the STAMP benchmarks. It only fails at a significant rate with bayes, intruder, and kmeans, as well as with labyrinth and yada but only when many threads are used. These numbers are consistent with the nature of the workloads: pre-validation is more likely to fail when conflicts with concurrent transactions are frequent. This optimization saves the cost of acquiring the master lock and slowing down the other threads.

For most benchmarks, pre-validation does not hold often. This is because helper threads typically request the master lock while the master thread is active. Once they succeed in acquiring it, the master has committed and incremented cntx, which requires helpers to validate again. Only for some STAMP applications (bayes, labyrinth, and kmeans) does pre-validation hold more than 10% of the time, and essentially in the case of 2 threads. These benchmarks have long-running transactions that reduce the likelihood of a concurrent commit.

Finally, we observe that the hand-over optimization is very effective in STAMP benchmarks that have workloads with transactions of different sizes, but less so for the intset benchmarks because transactions are short and identical, hence the master commits at a high rate and prevents helpers handling over the lock to another. As expected, hand-overs are generally more frequent when increasing the number of threads.
### 4.5 Communities of Interest Benchmark

In many applications, data can be naturally partitioned such that transactions operating on different partitions do not conflict and threads rarely access the same partition at the same time. Such applications can take full advantage of the FastLANE algorithm because each partition can conceptually have its own master. If several threads try to access a partition at the same time, all but the first one will do it speculatively as a helper.

To evaluate the benefits of FastLANE in such settings, we developed a new benchmark implementing an operator from the field of streaming and batch systems that calculates the communities of interest (COI) for communication networks [3, 29]. The COI operator processes telephone calls and calculates the callees that are most often called by a caller. Figure 6 schematically illustrates the operator. One transaction will merge a given sequence of calls (“window”) into the COI. The window has a configurable size \( W \) and is generated randomly before each transaction. The identities of callers and callees are randomly distributed between 1 and \( N \), and the duration of each call between 1 and 60 seconds.

The COI is computed for each caller that placed a call. The operator maintains a top-K list of size \( K = 9 \) in which callees are ordered by weight. The weight is calculated as a moving average with factor \( \theta \) (\( 0 \leq \theta \leq 1 \)): \( \text{weight} = \text{weight} \times (1-\theta) + \text{duration} \times \theta \). Thus, \( \theta \) defines how much a new record influences the data. As the transaction iterates over the window, it updates the weights of callees (possibly inserting new entries) and moves them in the right position in the top-K list.

The partitioning of the data is based on the identity of the caller in the COI. We extended FastLANE to support partitions in the following way: (1) each partition has its own master, cntr, and helpers; (2) the hash function to find the dirty entry was adapted to be aware of the partition and keeps disjoint partitions in dirty during the mapping; (3) at START each thread becomes master if there is no other master, and releases the master privilege on MASTER_COMMIT; (4) all optimizations are disabled. Note that the approach taken in this benchmark differs from related work on TM partitioning, e.g., [20], because we do not rely on tuning to determine which synchronization mechanisms to use in each partition, and we can benefit from more parallelism than a shared lock (a secondary thread can access the partition as a helper) and less overhead than multiple locks as used by many STMs.

Figure 5 shows the experimental results for FastLANE with partition support (FL-P) for different values of \( N \) (\( 2^{10} \) and \( 2^{15} \)) and \( W \) (1, 10, 20). FL-P is configured with 8 partitions. The window size essentially defines the size of a transaction, i.e., how many callees have to be merged.

A first observation is that, for \( W = 1 \), FL-P has a higher single-threaded overhead than FL because of the extra indirection to select the partition has an high impact with short transactions. Except for that configuration, FL-P performs and scales systematically better than FL and other STMs thanks to having one master per partition. If more threads are active than partitions (12 threads vs. 8 partitions), FL-P is still able to scale because the threads execute transactions as helpers. This contribution of helper threads is shown in Table 5, which lists the percentage of commits achieved by FL and FL-P for 3, 6, 8, and 12 threads. Without partitioning, the contribution of helpers grows from 30% to 55%. When using partitions, it remains in the 12% to 42% range because more threads execute as master, but it continues to grow when using more threads than partitions.

Note that using partitions with other STMs would not produce the same benefits as it would not suppress the overheads associated with transaction management (context saving, instrumentation).

### Table 5. Contribution of helpers to the global commit throughput.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Helper commits (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FL</td>
<td>FL-P</td>
</tr>
<tr>
<td>N=1024 W=01</td>
<td>41.8</td>
</tr>
<tr>
<td>N=1024 W=10</td>
<td>33.0</td>
</tr>
<tr>
<td>N=32768 W=10</td>
<td>32.9</td>
</tr>
<tr>
<td>N=32768 W=20</td>
<td>30.1</td>
</tr>
</tbody>
</table>

Table 6 shows the abort rates for all STMs and FastLANE when 12 threads are active. TinySTM and NOREC have very low abort rates that would not diminish much if one instance per partition would be used; only the contention on their shared counter would be reduced. TML has a very high abort rate because almost all transactions perform updates and hence are practically serialized. Having one instance per partition would correspond to having a shared counter per partition, enabling parallelism as long as not two threads try to access the same partition. FL has an abort rate that increases with large transaction sizes because the fast master will invalidate slow helper transactions more often. Finally, FL-P has a moderate abort rate due to the fact that more threads are active than partitions. As compared to having a shared lock per partition, the speculative helpers can execute concurrently with masters and allow for more parallelism.

One can notice a drop in the throughput of FL-P after 6 threads for \( W = 1 \) and, to a lesser extent, for \( W = 10 \). As previously said, this is due to fact that when more than 6 threads are active, the second socket of our experimental machine is in use, which creates expensive cache coherence traffic over the interconnect. For longer transactions (\( W = 20 \)), there is no noticeable degradation because the cache is less efficiently used even on a single processor.

Interestingly, FL-P provides significant improvements over sequential with 2 threads already, and scales remarkably well, making the switch to STM less crucial even for 12 threads (except for \( W = 1 \) where ETL wins starting with 8 threads).

As a final remark, this benchmark is representative of applications with low latency requirements, e.g., for stream processing. Having one master per partition allows us to guarantee fast processing with almost predictable execution times, comparable to non-instrumented execution.

### Table 6. Abort rates for STMs and FastLANE (12 threads).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Abort rates (%, 12 threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WT</td>
</tr>
<tr>
<td>N=1024 W=01</td>
<td>0.0</td>
</tr>
<tr>
<td>N=1024 W=10</td>
<td>1.5</td>
</tr>
<tr>
<td>N=32768 W=10</td>
<td>0.1</td>
</tr>
<tr>
<td>N=32768 W=20</td>
<td>0.6</td>
</tr>
</tbody>
</table>
5. Conclusion

In this paper, we have addressed one of the main drawbacks of STM: its limited performance at low thread counts, as compared to the execution of the sequential original application without the overheads of TM.

We have proposed a novel synchronization strategy, FASTLAME, designed to perform best at low thread counts, where classical STM implementations are slower than sequential execution—typically between 2 and 4 threads. FASTLAME relies on a single pessimistic master thread with light instrumentation that never aborts, and one or more speculative helper threads that perform additional work as they try to commit their transactions without hampering the progress of the master. We have presented several variants that differ mainly by the optimizations they introduce in the commit function of the helper threads.

We have introduced a new benchmark that computes communities of interest, with a transactional workload that can be partitioned such that multiple masters can execute in distinct partitions. Results from experimental evaluation show that our new algorithms performs well in their target operation range of low thread counts, and provide a real performance boost if combined with partitioning.

Currently, the decision of when to change between the different code paths (sequential, FASTLAME, and STM) is based on the number of physical cores available on the target machine when starting the application. In the future, we would like to investigate dynamic schemes by periodically switching modes for short durations to gather samples of the commit rate and taking decision point on that basis. This simple sampling-based approach could obviously be combined with more sophisticated strategies, e.g., using modeling-based techniques, depending on the workload nature.

References


