# FastLane: Streamlining Transactions for Low Thread Counts

Jons-Tobias Wamhoff
Christof Fetzer
Technische Universität Dresden, Germany
{jons,cf2}@inf.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 system 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 family of algorithms that differentiate between two types of threads: One thread (the master) is allowed to commit transactions without aborting, thus with minimal instrumentation and management costs, while other threads (the helpers) can commit 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 within a state-of-the-art STM runtime and compiler. Multiple code paths are produced for execution on a single, few, and many cores. Applications can dynamically select a variant at runtime, depending on the number of cores available for execution. Preliminary evaluation results indicate that our approach provides promising performance at low thread counts: FastLane almost systematically wins over a classical STM in the 2-4 threads range, and often performs better than sequential execution of the non-instrumented version of the same application. We believe that performance can still be improved by additional optimizations and tuning of the FastLane algorithms.

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

***General Terms*** Algorithms, Performance.

***Keywords*** Transactional Memory, Concurrent Programming.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*TRANSACT'12* February 26th, 2012, New Orleans, LA, USA.
Copyright © 2012 ACM . . . $10.00

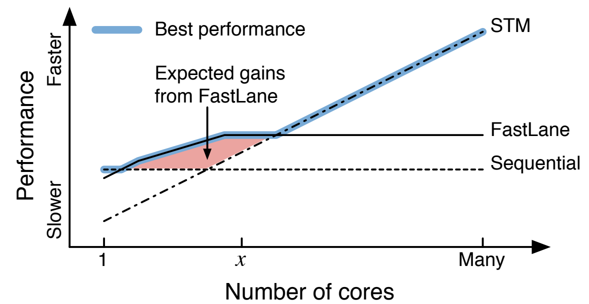


**Figure 1.** Our objective is to develop an algorithm that bridges the gap between sequential and STM performance at low thread counts.

## 1. Introduction

Transactional memory (TM) has been receiving much attention over the last decade as it provides a scalable and easy-to-use approach to concurrent programming. Developers simply enclose critical sections within transactions[1] that execute speculatively and abort when conflicting accesses to shared data are detected at runtime.

Most existing TM implementations are software-only as processors with dedicated hardware TM (HTM) instructions are not generally available yet. Therefore, our focus in this paper is on software TM (STM). STM implementations often exhibit excellent scalability with high thread counts [2, 4, 6, 7], but the overheads related to transaction management and instrumentation of memory accesses[2] are clearly visible when executing on few cores: the performance of a single-treaded non-instrumented application is generally higher than when using STM on a few cores [1–3, 9]. Only when reaching a certain threshold of cores (call it $x$) does STM pay off.

The objective of this work is to bridge the gap between single-threaded performance of the non-instrumented code and multi-threaded performance of the STM-based code on $x$ cores. To that end, we propose a family of novel algorithms, called FastLane, designed to execute as fast as possible on $1 < n \leq x$ cores. On that basis, one can develop applications that can best exploit the number of cores at their disposal by dynamically determining which algorithm to use: sequential for 1 core, FastLane for 2 to $x$ cores, or STM for more than $x$ cores. Figure 1 depicts schematically the expected behavior of the three execution strategies and the zone

[1] TM-aware compilers typically provide higher-level *atomic block* language constructs that are transparently mapped to transactions.

[2] Reads and writes to shared memory are replaced by transactional accesses, which trigger execution of complex operations (conflict detection, maintenance of the read/write sets, etc.) requiring hundreds of additional cycles.

where FASTLANE is expected to boost performance as compared to the state-of-the-art STM algorithms. Not that

The basic idea of FASTLANE is to have threads operate in one of two modes. One *master* thread runs at nearly sequential speed with only minimal instrumentation, while all other threads execute speculatively and try to help the master whenever they can. The latter threads, called *helpers*, typically run slower than STM threads, as they should not hamper progress of the master in addition to performing the extra bookkeeping associated with memory accesses. The roles of master and helper can change dynamically during execution of the concurrent application, although we did not use this functionality as part of our preliminary evaluation.

We propose several variants of the FASTLANE algorithm, which offer different tradeoffs in terms of complexity and performance. We expect the efficiency of the variants to depend mainly on the considered workload, e.g., level of contention, length of transactions, ratio of reads to writes, etc. We also present optimizations that can improve performance in certain conditions.

We generate multiple code paths for a given application: sequential, STM, and FASTLANE (master and helper). The choice of which variant to use is made when starting the application, depending on the amount of cores available on the target machine. While it would be conceptually possible to dynamically change the operating mode while the application is running, this would require stopping all threads and reconfiguring the runtime. Our current implementation does not support this functionality yet, but it has been shown in previous work that a dynamic tuning of the STM runtime system can significantly improve the throughput [10, 12, 15]. Here we focus on streamlining the STM algorithms for low thread counts.

We have evaluated the performance of the FASTLANE variants on a number of synthetic and realistic benchmarks from the STAMP suite [9], and compared them against STM and sequential executions. Although there is still room for improvement, our preliminary results show promising performance and we believe to make a case for having dedicated algorithms for the portion of the scale where STM loses against sequential execution, that is with low thread counts.

The rest of this paper is organized as follows. Section 2 describes the family of FASTLANE algorithms and explains the design choices that led to the different variants. Section 3 evaluates the performance of the algorithms on various synthetic and realistic benchmarks. Section 4 discusses related work and finally Section 5 concludes.

## 2. FASTLANE Algorithms

The high-level objective of the FASTLANE algorithms is to perform (1) no worse than sequential execution, and if possible (2) better when leveraging a few additional threads. To meet the first goal, we rely on a 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.

All the algorithms presented in this section follow this general approach, with variations in implementation details and optimizations. We start by describing the data structures and behavior of the master thread, before describing the different variants that mainly differ in the operation of the helper threads in their commit phase.

### 2.1 Data Structures

The data structures used by the FASTLANE algorithms are summarized in Table 1. They essentially consist of: a shared counter, `cntr`, that keeps track of the progress of the master and ensures mutual exclusion with helpers trying to commit their changes; a shared array of integers, `dirty[]`, that protects a set of memory addresses and stores the value of the counter at the last time one of these locations was updated; and a FIFO lock, `helpers`, to serialize commits of helpers.

| Variable | Description |
|---|---|
| `cntr` | Counter that tracks progress of the master (and, in some variants, of the helpers). The value is odd when the master is in a transaction, even otherwise. This variable is used to ensure mutual exclusion between the master and 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 `cntr` at the last time the address was written. |
| `helpers` | Lock to ensure mutual exclusion when validating transactions and serialize commits of helpers. It is implemented as a CLH list-based queue lock [8] and provides FIFO guarantees. |

**Table 1.** Shared data variables used by FASTLANE algorithms.

### 2.2 Master Thread

The operation of the master thread is described in Algorithm 1. Upon transaction start, the master increments the counter to an odd value. As the counter is also used by helpers to obtain exclusive access to shared memory during their commit phase, the master must first ensure that no helper is currently committing. This is achieved by checking if the counter is already odd (line 3). As other helpers may be waiting for the counter to become even, we want to give priority to the master. To that end, the master "reserves" the counter by atomically setting its most significant bit (line 2), which will result in the reservation by any helper (which is based on a compare-and-set operation) to fail. In practice, using the reservation instead of a compare-and-set operation will reduce the latency of the master. Once the master is guaranteed to have exclusive access to the counter, it simply increments it to the next odd value (line 5).

**Algorithm 1**: Master thread.

```
1  function START
2      atomic-or (cntr, MSB)                // Master request priority access
3      while (cntr & 0x01) ≠ 0 do           // Wait for committing helpers
4          wait
5      cntr ← (cntr & ¬MSB) + 1             // Only master can write odd cntr
6  function READ(addr)
7      return *addr                         // No instrumentation
8  function WRITE(addr, val)
9      dirty[hash(addr)] ← cntr             // Mark data as modified
10     addr ← val                           // No additional bookkeeping
11 function COMMIT
12     cntr++                               // Only master can write odd cntr
```

Instrumentation of memory accesses is minimal on the master thread. Reads are not instrumented (line 7), while writes are simply augmented by storing the value of the counter in the corresponding entry of the `dirty[]` array (line 9). Finally, upon commit, the master simply releases the counter by incrementing it to an even value (line 12). Note that, at this point, only the master can write the counter and thus we do not need to use an atomic operation.

As one can note from the algorithm, in most cases the master has very low overhead. Reads are not instrumented and writes go directly to memory. Two increments and *atomic or* operation to

**Algorithm 2**: Helper thread (common functions).

```
1  function START
2      start ← cntr & ∼1 & ¬MSB  // Get even counter (discard LSB & MSB)
3  function READ(addr)
4      if CONTAINS(write-set, addr) then      // Is address already in write-set?
5          return GET(write-set, addr)        // Return value previously written
6      val ← *addr
7      if dirty[hash(addr)] > start then                 // Validate read value
8          ABORT
9      ADD(read-set, addr)                          // Add address to read set
10     return val
11 function WRITE(addr, val)
12     if dirty[hash(addr)] > start then              // Validate write address
13         ABORT
14     PUT(write-set, addr, val)                // Add to (or update) write set
15 function EMITWRITESET                            // Apply writes to memory
16     foreach (addr, val) ∈ write-set do
17         dirty[hash(addr)] ← cntr
18         *addr ← val
19 function VALIDATE                          // Validate read and write sets
20     foreach addr ∈ read-set ∪ write-set do
21         if dirty[hash(addr)] > start then            // Concurrent update?
22             return false
23     return true
24 function WAITFOREVENCOUNTER             // Wait for counter to become even
25     repeat
26         c ← cntr
27     until (c & 0x01) = 0
28     return c & ¬MSB     // Return even counter value (ignore master flag)
```

*cntr* are performed per transaction, but this fixed overhead can be typically neglected for transactions of a certain duration. Finally, one update to the *dirty[]* array is necessary for every write. Bookkeeping of reads and writes is minimal, as transactions of the master thread never need to abort.

## 2.3 Helper Thread

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

The functions common to all FASTLANE helper variants are shown in Algorithm 2. Upon transaction start, we store the current value of the counter for subsequent validation purposes, discarding the most and least significant bits to force the value to be even (line 2).

When reading a memory location, the helper first checks whether it has already written that very address. If so, it returns the previously written address (lines 4–5). It then reads the value and conservatively checks it the address may have been concurrently written, by checking the associated entry of the *dirty[]* array; if so, the transaction simply aborts (lines 6–8). Otherwise, the read can successfully complete: the address is added to the read set and the previously read value is returned (lines 9–10).

Upon write, we check if the written address has possibly been updated concurrently, like for reads, and if so, the transaction aborts (lines 12–13). Otherwise, we simply add the address and the written value to the write set (line 14), delaying the actual update of the shared memory to the commit phase.

A few additional functions are used by the different FASTLANE helper algorithms. EMITWRITESET commits all updates stored in the write set to shared memory and updates the associated entries of the *dirty[]* array with the value of *cntr* (lines 15–18). VALIDATE verifies if any address stored in the read and write set may have been concurrently updated, by looking into the *dirty[]* array, and if so conservatively aborts (lines 19–23). Finally, WAITFOREVENCOUNTER waits until the shared counter *cntr* becomes even, i.e., appears to be available for acquisition (lines 24–28). The most significant bit, set by the master as a flag to obtain priority access to the counter, is discarded. The rationale behind using this function is that helper threads should not try to modify the counter in their commit phase with an atomic *compare-and-set* operation[3] unless it appears to be available. Otherwise, the contention on the cached *cntr* will significantly degrade performance.

**Algorithm 3**: Helper thread (commit variant 1).

```
1  function COMMIT_1
2      if EMPTY(write-set) then               //  Read-only transaction?
3          return                                  // Commit immediately
4      repeat                                       //  Try acquiring counter
5          c ← WAITFOREVENCOUNTER
6      until cas (cntr, c, c+1)   // Attempt C&S only after counter seen even
7      if ¬VALIDATE then
8          atomic-dec (cntr)      // Release counter upon failed validation
9          ABORT
10     EMITWRITESET                              // Write updates to memory
11     atomic-inc (cntr)                                   // Release counter
```

The first variant of the helper's commit function is shown in Algorithm 3. The main idea here is to perform the validation of the read and write sets, resulting in either an abort or a successful commit, while holding the counter. If the transaction is read-only (lines 2–3), all memory accesses have already been validated by the READ function and the transaction can commit immediately. Otherwise, the helper must ensure mutual exclusion for its commit phase. To that end, it first waits for the counter to appear available (i.e., have an even value) before attempting to lock it using a compare-and-set operation (lines 4–6). Note that the compare-and-set operation will fail if the master has set the MSB of *cntr* to request priority access. Finally, validation is performed while *cntr* has an odd value and no other thread (even the master) can interfere. Upon successful validation, pending updates in the write set are sent to memory (line 10). Finally, the counter is released by increasing it to the next even value (line 11). This operation must be performed atomically because the master might be concurrently setting the MSB of the counter.

The second variant is shown in Algorithm 4. The main difference with the previous version is that we try to first validate outside the critical section, i.e., without setting *cntr* to an odd value, and only acquire the counter if validation succeeds. This is expected to reduce contention with the master, as a transaction that is known to abort will not compete for the counter. We prevent multiple helpers from committing concurrently using the *helpers* FIFO lock in order to avoid interference with the pre-validation from other helpers (lines 4–18). Before validation, the helper first waits for *cntr* to have an even value, i.e., it appears to be available, and stores its value (line 5). It then proceeds to validation (line 6) and, if successful, tries to enter the critical section by setting *cntr* to the next odd value (line 10). Once in the critical section, if the value of the counter has not been modified by another thread since before validation, then we can commit directly; otherwise, we must

[3] The *compare-and-set* operation takes 3 parameters: an address, an expected value, and a new value. It atomically checks if the address contains the expected value and, if so, updates it with the new value and returns true; otherwise, the memory location is unmodified and the operation returns false.

**Algorithm 4**: Helper thread (commit variant 2).

```
1  function COMMIT_2
2      if EMPTY(write-set) then                          // Read-only transaction?
3          return                                        // Commit immediately
4      lock (helpers)                     // Only one helper at a time (FIFO lock)
5      c ← WAITFOREVENCOUNTER
6      if ¬VALIDATE then                   // Pre-validate before acquiring counter
7          unlock (helpers)               // Release lock upon failed validation
8          ABORT
9      t ← c+1                                       // Remember validation time
10     while ¬cas (cntr, c, c+1) do         // Likely commit: try acquiring counter
11         c ← WAITFOREVENCOUNTER
12     if cntr > t ∧ ¬VALIDATE then           // Check that validation still holds
13         atomic-dec (cntr)             // Release locks upon failed validation
14         unlock (helpers)
15         ABORT
16     EMITWRITESET                                    // Write updates to memory
17     atomic-inc (cntr)                                         // Release locks
18     unlock (helpers)
```

re-validate (line 12).[4] The writes of pending updates and exit of the critical section happen as for the previous variant.

**Algorithm 5**: Helper thread (commit variant 3).

```
1  function COMMIT_3
2      if EMPTY(write-set) then                          // Read-only transaction?
3          return                                        // Commit immediately
4      lock (helpers)                     // Only one helper at a time (FIFO lock)
5      repeat
6          c ← WAITFOREVENCOUNTER
7          if ¬VALIDATE then               // Pre-validate before acquiring counter
8              unlock (helpers)           // Release lock upon failed validation
9              ABORT
10     until c = cntr ∧ cas (cntr, c, c+1)
11     EMITWRITESET                                    // Write updates to memory
12     atomic-inc (cntr)                                         // Release locks
13     unlock (helpers)
```

The last variant, shown in Algorithm 5, performs validation entirely outside the critical section so that helpers interfere with the master only when they are guaranteed not to abort. The algorithm is adapted from [13]. As for the second variant, we serialize the commit phases of the helpers by means of a FIFO lock. Validation takes place in the loop of lines 5 to 10, which terminates only when validation and entry into the critical section both succeed in a row without $cntr$ being modified by another thread (line 10). The rest of the commit phase is similar to the second variant.

### 2.4 Irrevocability And Switching The Master Role

Transactions can also request at any time during their execution to become irrevocable. This requires the quiescence of all other threads. The requesting thread tries to acquire a quiescence lock that will stop all helper threads from starting new transactions. If the requesting thread is a helper and the quiescence lock is already taken, the transaction must abort. Otherwise the thread waits until no other transaction is active and then acquires the counter for priority access. Finally, the thread can continue with its irrevocable operations.

[4] We assume in the pseudo-code that the boolean operators *and* ($\wedge$) and *or* ($\vee$) use short-circuit evaluation, i.e., the second part of the condition is only evaluated if the first part does not suffice to determine the value of the expression.

Certain applications might require to change the role of a thread between master and helper. By default the first thread that executes a transaction will become the master. During the execution of the application any thread can request to become the master. Switching the roles of a thread is implemented in a quiescence period similar to irrevocability. First, all worker threads must be stopped and then, the requesting thread must acquire the counter using a compare-and-set operation. Finally, it can set itself as the master.

### 2.5 Optimizations

There are a number of possible optimizations to the algorithms presented above. We implement and evaluate the first one of these optimizations, and leave the other two as future work.

First, it is possible to reduce the number and cost of acquisitions of $cntr$ by the master thread, by letting helpers indicate when they need $cntr$ to be incremented. The master then only releases $cntr$ (i.e., increments it to the next even value) and subsequently reacquires it when there is at least one such pending request, registered in a global flag. The flag is set by a helper when (1) it needs to commit and must enter the critical section, or (2) the helper aborts because it could not validate. In the latter case, the helper needs the counter 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 worker that later reads the same variable will remember $x-1$ (even) as start value of the counter and will systematically fail validation until $cntr$ becomes greater than $x$ (see Algorithm 2, lines 2 and 21). This optimization is implemented in the second variant of the helper, which we name 2-opt in the evaluation section.

We also observe that, in variants 2 and 3 of the helper thread commit functions (Algorithms 4 and 5), access to the commit phase requires acquisition of the $helpers$ lock. Henceforth, at any point in time, at most two threads (one master and one helper) may try to simultaneously acquire $cntr$. It is thus possible to use a lighter synchronization mechanism that does not require atomic operations like compare-and-set (but will still incur the cost of cache invalidation).

Finally, it might be possible to extend the "pre-validation" principle further in order to detect doomed transactions as early as possible. Helper threads could try to validate during read and write, or before acquiring the $helpers$ lock or waiting for an even counter in the commit function. Obviously, as for variant 2, this requires re-validating once in the critical section of the commit function, but with a higher chance of successful validation.

## 3. Evaluation

In this section, we evaluate the performance of FASTLANE. We are specifically interested in showing that (1) it minimizes overhead for the master thread, (2) it scales for a low number of threads in a competitive manner against existing STM algorithms under little contention, and (3) it performs comparably to a sequential execution without instrumentation under high contention.

We compare FASTLANE against sequential execution and two variants of a state-of-the-art STM algorithms, TINYSTM [5, 6], 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.

For our evaluation, we use both the synthetic *intset* micro-benchmarks and realistic applications from the STAMP [9] benchmark suite. The *intset* micro-benchmarks perform randomly queries and updates on integer sets implemented as *red-black tree* (RB), *linked list* (LL), *skip list* (SL), and *hash set* (HS). From the STAMP benchmark suite, we chose `genome`, `intruder`, `kmeans`, `ssca2`, and `vacation`: `genome` performs gene sequencing using hash sets

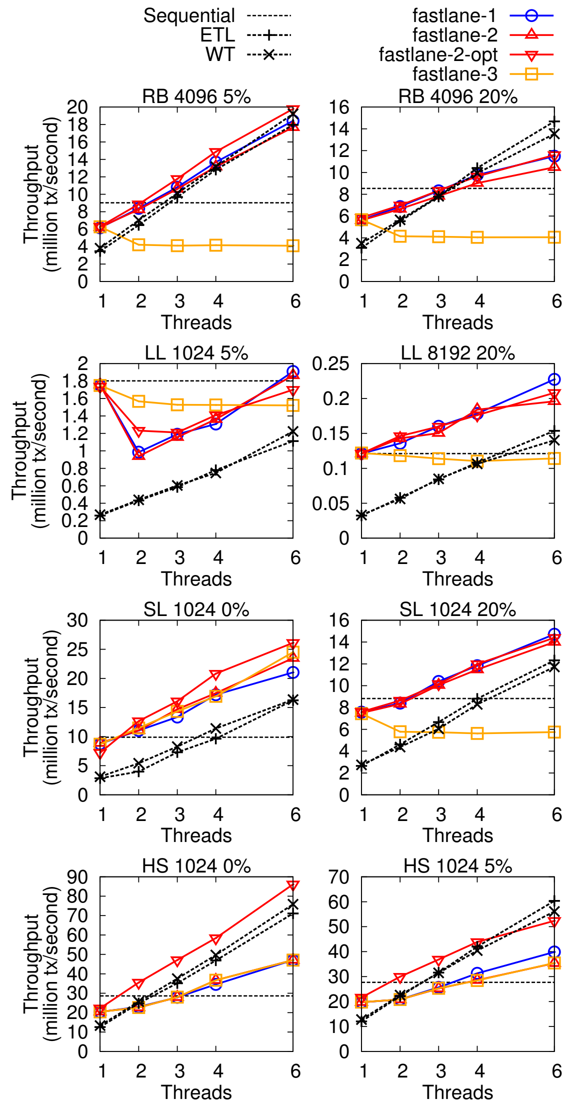


**Figure 2.** *Intset* benchmark: throughput (higher is better).

and string search; `intruder` emulates a signature-based network intrusion detection system by matching packets against signatures stored in self-balancing trees; `kmeans` clusters a set of partitioned points in parallel; `ssca2` constructs an efficient graph data structure using adjacency arrays; finally, `vacation` emulates a travel reservation system, reading and writing different tables that are implemented as red-black trees.

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. We compiled all micro-benchmarks with manual transactional instrumentation using GCC 4.6 and enabled link-time optimization. The CPU affinity was configured such that the penalty of moving data between sockets is as limited as possible. All six cores of a processor share the L3 cache. When up to 6 threads are active, which is the case in our experiments with the *intset* micro-benchmarks, only one of the two processors is used.

We consider the following variants of the *intset* micro-benchmarks. For RB, we use a working set of 4,096 elements with update-to-lookup ratios of 5% and 20%; the working set is 1,024 elements for the other variants, with update ratios of 5% and 20% for LL, 0% (read-only) and 20% for SL, and 0% and 5% for HS. We present in Figure 2 the throughput (transactions committed per second computed during a period of 10 seconds of execution), with the sequential execution (non-TM) throughput as a baseline and represented by a horizontal dashed line. Figures 3, 4, 5, and 6 detail the numbers of commits and aborts for a subset of the *intset* micro-benchmarks, representative of various conditions for contention and transaction length. We distinguish between the commits of the master, and the commits and aborts of the helper threads. When using more than one helper, we present the *average* number of commits, respectively aborts, over the execution period for *all* helpers. Our last set of experiments, presented in Figure 7, evaluate the throughput of STAMP applications. We selected all benchmarks that were compatible with FASTLANE. The benchmark were configured accordingly to the documentation with high contention parameters.

### 3.1 Single Thread Overhead

Our main goal is to reduce the overhead introduced by synchronization for low thread counts. The single-threaded throughput is usually the base that can give subsequent threads a head start. Therefore, we first focus on the single threaded overhead, that is, when only processing transactions on the master thread. The operation of the master is lightweight: it only needs to increment the counter upon beginning and committing a transaction. Loads have no instrumentation at all, while writes only require an additional update to the `dirty[]` array.

On the other end of the spectrum, state-of-the-art STM algorithms require non-trivial algorithms to be executed for every transactional operation. They must typically copy the current CPU context at transaction *begin* to support restart upon abort, keep track of read and write sets upon memory accesses, and perform validation and memory copy operations upon commit. FASTLANE's objective is to streamline these costs for the master thread.

Figures 2 and 7 show that the master can indeed achieve single-threaded throughput close to that of sequential execution. For *intset*, the performance is very close to sequential for LL, SL, `genome`, `intruder`, and `ssca2`. While still better than other STM, the performance for RB and HS is not as close to sequential: this is primarily due to the fact that these benchmarks have small transactions, hence the cost associated with the management of `cntr` in the begin and commit phases dominates.

Note that the single threaded overhead could be further reduced. FASTLANE provides a generic interface that checks at each invocation of the begin, read, write, and commit functions if it is currently a master or helper thread. In the *intset* benchmarks LL, SL, and HS in Figure 2, we manually differentiated the code paths of the master and helpers at the application level, imitating efficient compiler-generated code. The RB benchmark, which has more complicated application logic, uses generic variants of the transactional operations that execute additional branches at runtime, hence incurring overheads that could be avoided with smarter compiler support. These compiler-side optimizations are part of our ongoing work.

### 3.2 Scalability for Low Thread Counts

We now want to show that the helper threads allow the FASTLANE variants to scale for low thread counts. Our design streamlines the master for minimal overhead. Helpers have the overhead of bookkeeping the transactional metadata and validation at commit time. This results in a very unbalanced workload distribution between the two types of threads because the master is able to process transactions much faster than the workers. Therefore we adapted the STAMP benchmarks with a partitioning-based dynamic work balancing that introduces only very little overhead. For the *intset*

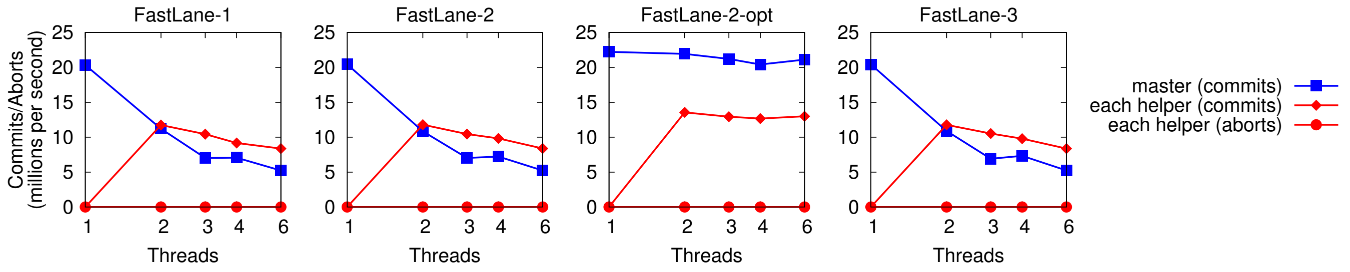


**Figure 3.** *Intset* benchmark: commits and abort rates for the master and helpers (HS, 1024 elements, no updates).

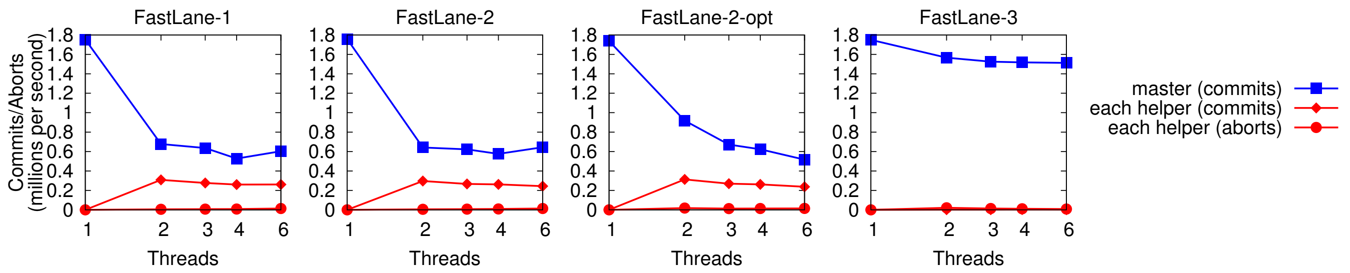


**Figure 4.** *Intset* benchmark: commits and abort rates for the master and helpers (LL, 1024 elements, 5% updates).

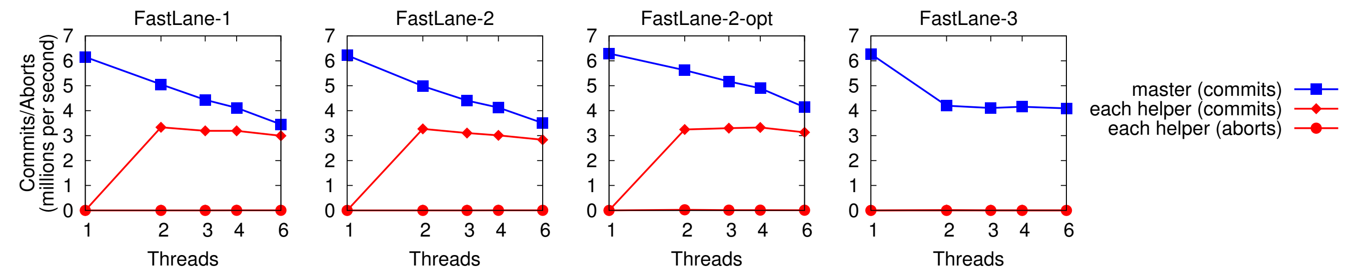


**Figure 5.** *Intset* benchmark: commits and abort rates for the master and helpers (RB, 4096 elements, 5% updates).

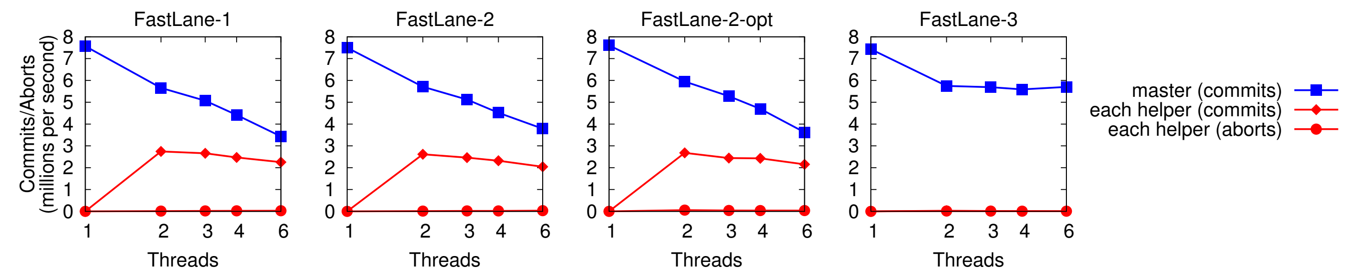


**Figure 6.** *Intset* benchmark: commits and abort rates for the master and helpers (SL, 1024 elements, 20% updates).

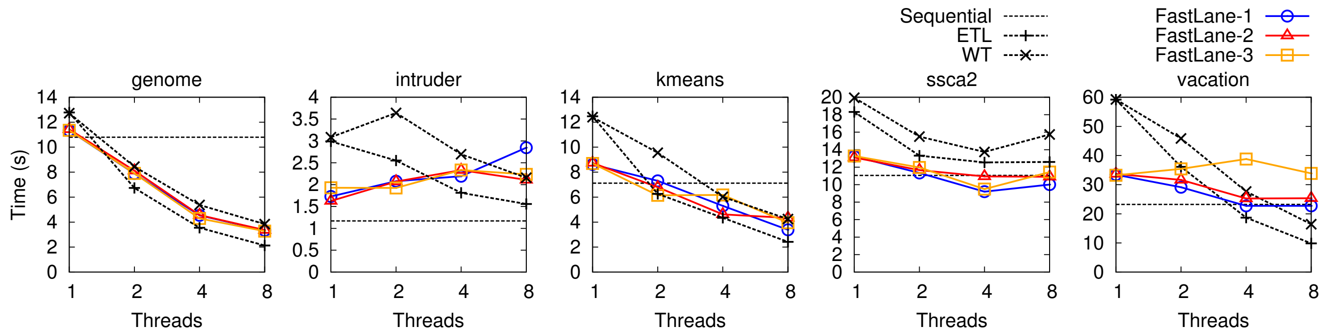


**Figure 7.** Completion times of four of the STAMP benchmarks [9]: `genome`, `kmeans`, `intruder`, `ssca2`, `yada` and `vacation` (all with high contention configuration, lower is better).

micro-benchmarks, each thread executes in a loop at its own speed for a given duration.

Overall, FASTLANE scales well for low thread counts with moderate contention, as shown in Figures 2 and 7. The minimal overhead of the master thread gives it a head start and the helpers contribute their share when the number of threads increases (see Figures 3, 4, 5 and 6). The abort rates are very low. The sequential execution is often outperformed with just two threads. Depending on the contention of the workload, TINYSTM wins over the other variants when more threads are added and the underlying algorithms exploit sufficient parallelism.

We observe that FASTLANE variant 3 does not scale well because the helpers cannot contribute to the throughput. This is due to the fact that almost all time is spent processing transactions in the *intset* benchmarks. Helpers need a realistic chance to validate outside the critical section (when holding $helpers$ and $cntr$), otherwise they simply cannot contribute to the overall performance. This is not the case in the considered micro-benchmarks as the master keeps updating $cntr$ at a high frequency and, hence, the value of the counter almost always changes between validation and the *compare-and-set* operation at the helpers. Therefore, there is a trade-off between possibly slowing down the master and creating a bottleneck for the helpers. Note that once the helper commits, it might perform a number of read-only transactions without interference with the master depending on the update rate, a share of work that would be lost if the helper is constantly blocked.

The HS benchmark has very short transaction, thus, the fixed transaction costs upon begin and commit have an higher impact. The plots for 0% update in Figure 2 show that the contention on the counter has a negative impact on scalability. The reason is cache contention, as un update of the counter by the master will invalidate the associated cache line in the cores executing the helper algorithm. Figure 3 compares the commits of the master vs. helper and shows the slowdown of the master. The increasing number of helpers prevents the master from contributing as much to the overall throughput.

Finally, Figures 2 and 3 show that the optimized variant 2 can keep the lead when adding more helper threads. The master thread does not increment the counter unless it is explicitly requested by a helper. For 0% updates, the counter is never incremented and all other variants are outperformed.

### 3.3 Small Penalty on Contention

High contention workloads typically yield bad scaling with state-of-the-art STMs because of high abort rates. With FASTLANE, we want to minimize the influence of helper threads on the master thread in order to keep the throughput close to the level of the single threaded execution. The helpers should only have an positive impact on the throughput when they are able to successfully commit. Variant 3 has the benefit of performing all validation without incrementing the counter itself, and never aborts once it does increment the counter. We expected to see a similar behavior with variant 2, but the single pre-validation is not sufficient to limit the impact on the master thread (see Figures 3, 4, 5 and 6).

## 4. Related Work

A variety of efficient software transactional memory implementations have been proposed in the last few years [2, 4, 6, 7, 14]. The main focus was on exploiting the available disjoint access parallelism with high thread counts. The best performing algorithms typically use revocable locks and time-based validation. Designated STM runtime systems [10–12, 15] reduce the bookkeeping overhead when no contention is present.

We are only aware of a few existing STM designs that target small thread counts. Transactional mutex locks (TML) [3] use a versioned reader-writer lock: read-only transactions can concurrently execute and commit but, as soon as a transaction wants to write, it must acquire the lock, which will lead to an abort of all other active transactions. While no other transaction can execute concurrently when an update transaction is active, the benefit is a minimal instrumentation overhead. Transactions only have to save the context upon start to support retries. No write or undo logs are needed, and transactional loads only have to check the status of the versioned reader-writer lock. The scalability with write-dominated workloads is obviously limited by this approach.

NOREC [4] extends the idea of TML with value-based validation to deal with concurrent updates. This allows read transactions to execute concurrently with update transactions. While ownership records are still omitted, maintenance of a read-set adds overhead compared to sequential execution of the code.

One should finally note that, in our previous work on the ROBUSTM [16] transactional memory, we already used the *atomic or* to break *compare-and-set* loops in order to allow a privileged thread to steal locks from other transactions. Using this mechanism, we implemented a practically wait-free STM algorithm that tolerates crashes and non-terminating transactions.

## 5. Conclusion

Until HTM becomes generally available in multi-core processors, STM will continue to be the dominant form of transactional memory. 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 original application on a single core without the overheads of TM.

We have proposed a family of new STM algorithms, FASTLANE, designed to perform best at low thread counts, where classical STM implementations are slower than sequential execution—typically between 2 and 4 threads. FASTLANE relies on a single master thread with light instrumentation that never aborts, and one or more 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 in the commit function of the helper threads.

Our next step will be an extension of the STM runtime and compiler support. The DTMC transactional C/C++ compiler already generates binaries with multiple code paths corresponding to different algorithms tailored for various numbers of core: sequential execution on a single core, FASTLANE on few cores, STM on many cores. The dynamic choice of the code path to execute is then driven by the load or number of available cores on the target machine. Results from our preliminary evaluation show promising results, but also tends to indicate that there is still room for improvement.

## References


[1] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of Eurosys'10*, Paris, France, apr 2010.

[2] O. S. D. Dice and N. Shavit. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, Stockholm, Sweden, 2006.

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

[4] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, Bangalore, India, 2010.

[5] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 237–246, Salt Lake City, UT, USA, 2008. ACM.

[6] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, dec 2010.

[7] S. M. Fernandes and J. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proc. of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 179–188, San Antonio, TX, USA, 2011.

[8] P. S. Megnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proc. of the 8th International Symposium on Parallel Processing*, IPPS, Cancon, Mexico, April 1994.

[9] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi- processing. In *Proc. of the IEEE International Symposium on Workload Characterization*, IISWC 2008, pages 35–46, Seattle, WA, USA, sep 2008.

[10] T. Riegel, C. Fetzer, and P. Felber. Automatic data partitioning in software transactional memories. In *20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2008.

[11] M. Spear, K. Kelsey, T. Bai, L. Dalessandro, M. Scott, C. Ding, and P. Wu. Fastpath speculative parallelization. In G. Gao, L. Pollock, J. Cavazos, and X. Li, editors, *Languages and Compilers for Parallel Computing*, volume 5898 of *Lecture Notes in Computer Science*, pages 338–352. Springer Berlin / Heidelberg, 2010.

[12] M. F. Spear. Lightweight, robust adaptivity for software transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 273–283, New York, NY, USA, 2010. ACM.

[13] M. F. Spear, M. M. Michael, and C. von Praun. Ringstm: Scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 275–284, New York, NY, USA, 2008. ACM.

[14] J. Sreeram, R. Cledat, T. Kumar, and S. Pande. RSTM: A relaxed consistency software transactional memory for multicores. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, page 428, Washington, DC, USA, 2007.

[15] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 3 –14, sept. 2009.

[16] J.-T. Wamhoff, T. Riegel, C. Fetzer, and P. Felber. RobuSTM: a robust software transactional memory. In *Proceedings of the 12th international conference on Stabilization, safety, and security of distributed systems*, SSS'10, pages 388–404, 2010.