FASTLANE
Improving Performance of Software Transactional Memory for Low Thread Counts

Jons-Tobias Wamhoff
Christof Fetzer
Technische Universität Dresden, Germany

Pascal Felber
Etienne Rivière
Université de Neuchâtel, Switzerland

Gilles Muller
INRIA, France
Transactional Memory (TM)

- **Concurrent programming** necessary for speedup
  - Multi-cores are everywhere
- **Synchronization** of parallel threads on shared memory challenging using locks or non-blocking primitives
  - Scalability, deadlocks, composability, conventions, ...
- **TM** can simplify concurrent programming
  - Execute sequence of instructions atomically:
    - ```
      __transaction { a = a - x; b = b + x; }
    ```
  - Hardware TM (HTM): fast performance
    - limited capacity, not available on all platforms
  - Software TM (STM): fallback library for HTM
    - scalable but typically high overhead
STM Challenges

<table>
<thead>
<tr>
<th>Performance</th>
<th>Faster</th>
<th>Slower</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>One</td>
<td>Many</td>
</tr>
</tbody>
</table>

- **STM**
- **FastLane**
- **Sequential**

Expected gains from FastLane:
- Absolute performance loss due to bookkeeping
- Relative performance improves with #cores
FASTLANE: General Idea

- 1 pessimistic master thread
  - Commits transactions without aborting
  - Minimal instrumentation and bookkeeping
  - Runs almost at sequential non-instrumented speed
- N speculative helper threads
  - Commit transactions only when not in conflict
  - Contribute progress without impairing on the performance of the master
  - Typically slower than STM threads
Multiple Code Paths

- Dresden TM Compiler generates code paths with different TM algorithms
  - Transforms \texttt{__transaction\{\}}: code uses TM for memory accesses
  - Generic \texttt{START} and \texttt{COMMIT} calls with internal branch
  - READ and WRITE are specific to code path and inlined
- TinySTM++ TM runtime
  - Dynamically select code path based on core/thread count at \texttt{START}

![Diagram showing code paths and execution flow]

Christie et al.: Evaluation of AMD’s Advanced Synchronization Facility Within a Complete Transactional Memory Stack, EuroSys ’10
FASTLANE: Metadata

**Counter (cntr)**
odd: transaction updates
even: otherwise

**Master Lock (mstr)**
TTAS lock
protects shared variables

**Helpers Lock (hlpr)**
MCS queue lock (FIFO)
serialize helper commits

**Memory**

- Address written
- Address read
- Address read

**Dirty array (dirty[])**

- Timestamp (odd)
- Timestamp (odd)
- ... (multiple timestamps)

**Master thread**

- isMaster

**Helper thread**

- Start timestamp (even)
- Write-set
- Read-set
- ...
**FASTLANE: Algorithm**

**MASTER**

- **START**
  - `ttas-lock(mstr)`

- **READ (addr)**
  - return `*addr`

- **WRITE (addr, val)**
  - if `¬(cntr & 0x01)` `cntr++`
  - `dirty[hash(addr)] = cntr`
  - `*addr = val`

- **COMMIT**
  - if `¬(cntr & 0x01)` `cntr++`
  - `ttas-unlock(mstr)`

**HELPER**

- **START**
  - `start = cntr & ~1`

- **READ (addr)**
  - `dirty[hash(addr)] ≤ start`
  - `add(read-set, addr)`
  - return `*addr`

- **WRITE (addr, val)**
  - `dirty[hash(addr)] ≤ start`
  - `put(write-set, addr, val)`

- **COMMIT**
  - `¬empty(write-set)`
  - `mcs-lock(hlpr) ∧ ttas-lock(mstr)`
  - `validate(read-set ∪ write-set)`
  - `update(write-set, ++cntr) ∧ cntr++`
  - `ttas-unlock(mstr) ∧ mcs-unlock(hlpr)`
Overheads: Master vs. Sequential

- Once \textit{mstr} is acquired using TAS instruction
  - No access to transaction descriptor required
  - \texttt{READ} not instrumented
  - Exclusive update of \texttt{dirty[]} using \texttt{cntr} upon each \texttt{WRITE}
  - \texttt{cntr} is cached after increment and cannot be invalidated
  - No atomic operations or barriers required
- \textit{mstr} release serializes transaction‘s updates
Performance

STAMP Vacation

- read- & write-set
- validation
- malloc/free mgmt
- save context
- validate reads
- helper commits

Felber et al.: Dynamic Performance Tuning of word-based Software Transactional Memory, PPoPP '08

Dalessandro et al.: Transactional Mutex Locks, Euro-Par '10

Dalessandro et al.: NOrec: Streamlining STM by Abolishing Ownership Records, PPoPP '10

Cao Minh et al.: STAMP: Stanford Transactional Applications for Multi-Processing, IISWC '08
Performance

- Master overhead depends on transaction size and number of WRITEs
- Helpers contribute for scalability if not in conflict
- Increased contention can slow down master

### Throughput (million txns/sec)

**Linked List:** 2048 elem, 5% upd

**Hash Set:** 1024 elem, 5% upd

**Helper commits:**
- 16%
- 60%
- 37%
- 78%
Communities of Interest Benchmark

- **Workload**: Operator from streaming system with partitioning
  - Find most frequent callees of a caller
  - Weight calculated as moving duration average with factor $\theta$
- **Transaction**: Merge Window into Top-K
  - Update weight or replace entry and sort Top-K
  - Partitions based on CallerID: each with $cntr$, $mstr$, $hlpr$

![Diagram](Diagram.png)
Communities of Interest Benchmark

- 8 partitions, first entering thread becomes master

<table>
<thead>
<tr>
<th>Helper</th>
<th>3 thr</th>
<th>6 thr.</th>
<th>12 thr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fastlane</td>
<td>42 %</td>
<td>55 %</td>
<td>49 %</td>
</tr>
<tr>
<td>FastLane-P</td>
<td>14 %</td>
<td>28 %</td>
<td>43 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Num=1024 Win=1</th>
<th>3 thr</th>
<th>6 thr.</th>
<th>12 thr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>33 %</td>
<td>50 %</td>
<td>55 %</td>
</tr>
<tr>
<td>TinySTM</td>
<td>12 %</td>
<td>24 %</td>
<td>40 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Num=32768 Win=10</th>
<th>3 thr</th>
<th>6 thr.</th>
<th>12 thr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>3 %</td>
<td>0 %</td>
<td>7 %</td>
</tr>
<tr>
<td>TinySTM</td>
<td>12 %</td>
<td>24 %</td>
<td>40 %</td>
</tr>
</tbody>
</table>

- 8 partitions, first entering thread becomes master

![Graph showing throughput for different scenarios](image-url)
FASTLANE: Optimizations

1. **Keep-lock** (FL-HO)
   - Master keeps $mstr$ if no helper requests it
   - Avoid unnecessary updates to shared variables

2. **Pre-validate** (FL-PV)
   - Helpers validate before attempting to acquire $mstr$
   - Stop master only if a high likelihood of successful commit

3. **Hand-over** (FL-HO)
   - Helpers can hand over $mstr$ if a successor in $hlpr$ queue
   - Increase commit chance
Hand-Over Performance

- Master keeps lock or helper passes lock on
- Increases chance for helpers to commit
- Reduces contention on $mstr$ and $cntr$

Cao Minh et al.: STAMP: Stanford Transactional Applications for Multi-Processing, IISWC '08
Pre-Validate Performance

- Reduces impact of helpers on master

Cao Minh et al.: STAMP: Stanford Transactional Applications for Multi-Processing, IISWC ’08
Conclusion

• STM has limited performance at low thread counts
• Pessimistic master thread with light instrumentation runs close to sequential speed
• Speculative helper threads provide scalability for low thread counts
• Partitioned workloads have multiple master threads, each saving the high STM overhead
Thank you!
Backup Slides
Irrevocable Operations

- Pessimistic code paths (sequential or master) used for irrevocable operations
  - **Sequential** path requires quiescence lock:
    - Stops all other transactions
  - **Master** path can be dynamically selected:
    - Helper aborts and acquires \textit{mstr} upon restart
    - Setting \textit{mstrID} to thread ID reflects new master
    - Other helpers can execute concurrently
Code Path Selection

function START(pessimistic)
  if masterID = threadID then
    ttas-lock(master)
    if masterID = threadID then
      MASTERSTART
      return CP_MASTER
    else
      ttas-unlock(master)
      HELPERSTART
      return CP_HELPER.
  else if pessimistic then
    ttas-lock(master)
    masterID ← threadID
    return CP_MASTER
  else
    HELPERSTART
    return CP_HELPER.
Master Code Path

\[
\text{function } \text{MASTER\_START} \\
\text{return } \text{MASTER\_START} \\
\text{function } \text{MASTER\_READ}(addr) \\
\quad \text{return } *addr \\
\text{function } \text{MASTER\_WRITE}(addr, val) \\
\quad \text{if } \neg(cntr \& 0x01) \text{ then} \\
\quad \quad \text{cntr } \leftarrow cntr + 1 \\
\quad \quad \text{dirty}[\text{hash}(addr)] \leftarrow cntr \\
\quad \quad addr \leftarrow val \\
\text{function } \text{MASTER\_COMMIT} \\
\quad \text{if } cntr \& 0x01 \text{ then} \\
\quad \quad \text{cntr } \leftarrow cntr + 1 \\
\quad \quad \text{ttas-unlock}(master) \\
\]
**Helper Code Path**

```plaintext
function HelperStart

  setjmp (ctxt)
  start ← cntr & ~1

function HelperRead(addr)

  if CONTAINS(write-set, addr) then
    return GET(write-set, addr)

  val ← *addr
  if dirty[hash(addr)] > start then
    ABORT

  ADD(read-set, addr)
  return val

function HelperWrite(addr, val)

  if dirty[hash(addr)] > start then
    ABORT

  PUT(write-set, addr, val)

function Validate

  if cntr ≤ start then
    return true

  foreach addr ∈ (read-set ∪ write-set) do
    if dirty[hash(addr)] > start then
      return false

  return true

function HelperCommit

  if EMPTY(write-set) then
    return

  mcs-lock (helpers)
  ttas-lock (master)
  if ¬VALIDATE then
    ttas-unlock (master)
    mcs-unlock (helpers)
    ABORT

  cntr ← cntr + 1
  foreach (addr, val) ∈ write-set do
    dirty[hash(addr)] ← cntr
    *addr ← val

  cntr ← cntr + 1
  ttas-unlock (master)
  mcs-unlock (helpers)

function Abort

  CLEAR(read-set, write-set)
  longjmp (ctxt)
```
Intset Benchmarks

Figure 3. Throughput of the Intset benchmarks (higher is better).

Figure 4. Completion times of the STAMP benchmarks [17] (all with high contention configuration, lower is better).

overhead for the FASTLANE 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 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 intset, 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 kmeans 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
STAMP Benchmarks

Figure 3. Throughput of the Intset benchmarks (higher is better).

Figure 4. Completion times of the STAMP benchmarks [17] (all with high contention configuration, lower is better).

- **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 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 intset, 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 kmeans 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
COI Benchmark