# Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory

Irina Calciu
Brown University
irina@cs.brown.edu

Tatiana Shpeisman
Gilles Pokam
Intel Labs
{tatiana.shpeisman, gilles.a.pokam}@intel.com

Maurice Herlihy
Brown University
mph@cs.brown.edu

## Abstract

Intel's Haswell and IBM's Blue Gene/Q and System Z are the first commercially available systems to include hardware transactional memory (HTM). However, they are all best-effort, meaning that every hardware transaction must have an alternative software fallback path that guarantees forward progress. The simplest and most widely used software fallback is a single global lock (SGL), in which aborted hardware transactions acquire the SGL before they are re-executed in software. Other hardware transactions need to subscribe to this lock and abort as soon as it is acquired. This approach, however, causes many hardware transactions to abort unnecessarily, determining even more transactions to fail and resort to the SGL.

In this paper we suggest improvements to the simple SGL fallback. First, we use lazy subscription to reduce the rate of SGL acquisitions. Next, we propose fine-grained conflict detection mechanisms between hardware transactions and a software SGL transaction. Finally, we describe how our findings can be used to improve future generations of HTMs.

***Categories and Subject Descriptors*** D [*1*]: 3

***General Terms*** Algorithms, Design, Performance

***Keywords*** Hardware Transactional Memory, Single Global Lock Fallback, Haswell RTM

## 1. Introduction

Parallel programming has gained significant importance due to the rise of commodity multicore computer systems. Unfortunately, writing correct and efficient software that effectively utilizes the resources of multicore systems remains an obstacle for a wider spread adoption of parallel programming. Locks, the current state-of-the-art solution for synchronizing shared memory access in parallel programs, are notoriously challenging, even for expert programmers [14].

Transactional memory (TM) is a synchronization paradigm [13] that aims to simplify writing correct and efficient parallel software while avoiding the pitfalls of locks. Threads can speculatively execute transactions, the synchronization primitive for TM, maintaining read and write sets to track conflicts. If a conflict is detected between two transactions, one is usually aborted and rolled back so the other can commit. Transactional memory generally promises all-or-nothing semantics, where critical sections appear as if they executed atomically or not at all. There have been many software transactional memory (STM) proposals [1, 2, 6, 10, 12, 19, 20, 22], where TM is implemented entirely in software. A specific example of such an STM is TL2 [9], a lock-based STM that we use in our study. Unfortunately, the overhead associated with these designs is generally prohibitive, making them unrealistic for production quality parallel software.

Hardware transactional memory (HTM), on the other hand, promises a faster performing, lower overhead alternative to STM. Yet, practical HTMs are *best-effort*: they do not guarantee forward progress. Furthermore, practical HTMs are bounded in size and support a restricted set of operations. It is for these reasons that an HTM alone is an insufficient TM solution.

In short, ensuring forward progress requires a software fallback. *Hybrid transactional memory* (HyTM) allows unsuccessful hardware transactions to revert to software, [5, 7, 8, 17, 18, 21, 23]. Existing HyTM proposals have drawbacks: memory accesses must be annotated, duplicate code is required for transactional and non-transactional paths, and it is not easily applied to legacy code. For many applications, however, it is simpler and more attractive to use a *single global lock* (SGL) mechanism [23, 24], where all transactions that access a particular data structure synchronize through a single lock[1]. Perhaps the most visible example of an SGL fallback scheme is Haswell's *hardware lock elision* (HLE) [15], which supports a lock fallback directly through the instruction set architecture. SGL schemes are attractive because they can easily be retrofitted to legacy code, and they do not require code duplication.

In both HLE, and HTM with SGL fallback, each hardware transaction starts by reading the lock's state, called *subscribing* to the lock. Subscription ensure that any software transaction that subsequently acquires that lock will provoke a data conflict, ensuring correctness by forcing any active subscribing hardware transactions to abort. The duration of a lock subscription represents a "window of vulnerability" during which the arrival of a software transaction will prevent any subscribing hardware transactions from executing.

In this paper we present novel optimizations to the simple SGL fallback approach. We show that one can significantly improve performance by performing lock subscription in a *lazy* manner: optimistically postponing reading the lock state for as long as possible (usually the very end of the transaction). Lazy subscription was first proposed in the context of Hybrid NOrec [5] to allow concurrent execution of multiple hardware transactions with the committing phase of a speculative software transaction. Here, lazy subscription allows concurrent execution of multiple hardware transactions with a single non-speculative SGL transaction. The resulting mechanism maintains the simplicity and correctness of the original SGL fallback, but reduces its costs. We evaluate this design using Haswell's *restricted transactional memory* (RTM) running the STAMP benchmark suite, and compare it to several alternatives:

[1] "Global" here could mean a single lock per data structure, not necessarily system-wide if composability is not an issue.

a non-speculative SGL implementation, a speculative implementation with the usual SGL fallback, the hardware-only HLE, and to an STM (TL2). We also show how to improve conflict detection with the SGL transaction and we propose several novel hardware extensions.

This paper makes the following contributions:

1. We propose an alternative implementation of the commonly used SGL fallback, using lazy subscription, that reduces the rate of lock acquisitions on Haswell and evaluate it on the STAMP benchmark suite.
2. We describe novel conflict detection algorithms between hardware transactions and an SGL software transaction.
3. Based on our experience, we suggest some simple optimizations to hardware lock elision mechanisms such as Haswell's.

## 2. SGL Fallback (E-SGL)

As noted, hardware transactional memory (HTM) has become a commercial reality, but HTM provided by processors such as Intel's Haswell and IBM's Power ISA offer no progress guarantees, implying that some form of software fallback is needed. In the single global lock (SGL) approach, each shared data structure has an associated lock. When it starts, a hardware transaction immediately reads the lock state, an action known as *eager subscription*. When a repeatedly failed hardware transaction restarts in software, it acquires exclusive access to the lock, forcing any subscribed hardware transactions to abort.

SGL fallback is attractive because it is simple, requiring no memory access annotation, and no code duplication between alternative paths. Nevertheless, an inherent limitation of current SGL fallbacks schemes is that hardware and software transactions that share a global lock cannot execute concurrently. Figure 1 shows the four ways in which hardware and software transaction can overlap. In cases 2 and 3, the hardware transaction is aborted as soon as it checks the lock, while in cases 1 and 4 the hardware transaction is aborted when the software transaction acquires the lock. With eager subscription, it makes sense for a thread starting a hardware transaction to wait until the SGL becomes free.

In this paper, we describe how to improve conflict detection to allow some concurrency between the hardware and software transaction that share a lock. In Section 3, we describe a *lazy subscription* mechanism that permits concurrent hardware and software transactions to share the same SGL and intuitively show its correctness. We evaluate this scheme's performance in Section 4. We describe finer-grained conflict detection mechanisms in Section 5. In Section 6, we describe how these observations might improve future hardware.

## 3. Lazy SGL (L-SGL)

In a naïve SGL implementation (E-SGL), a hardware transaction immediately adds the lock to its read set, ensuring the transaction will be aborted if that lock is acquired by a software transaction. Hardware and software transactions cannot overlap (Figure 1).

Lazy subscription can improve the chances of success of a hardware transaction by allowing some overlap with a software transaction. In Figure 2, L-SGL allows transactions (3) and (4) to commit, while E-SGL would abort them.

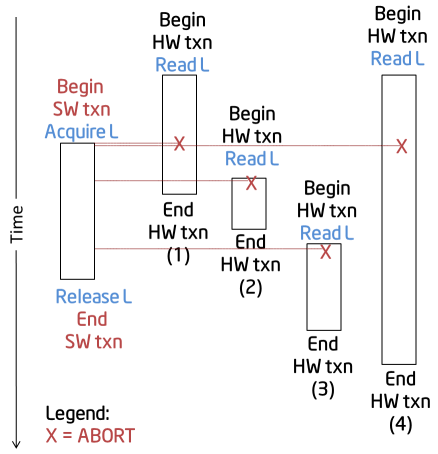



**Figure 1.** Obvious SGL Fallback implementation (E-SGL).

Software and hardware transactions are treated differently in L-SGL. Each software transaction must acquire the SGL. Hardware transactions do not acquire the SGL, but they must check its status. With some exceptions described later, L-SGL hardware transactions read the lock only at the *end*, right before committing. If the lock is held by a software transaction, the hardware transaction explicitly aborts. This check is necessary because the hardware transaction may have observed an inconsistent state. If the lock is free, then no software transaction is in progress, and the hardware transaction can commit.

Lazy subscription has been proposed to improve HyTM performance [5], but its use for SGL fallback is new. HyTMs typically use sophisticated techniques to allow concurrency between multiple hardware and software transactions, but SGLs' simplicity makes them attractive in practice [23, 24]. The lazy SGL (L-SGL) approach described here improves a popular HTM fallback mechanism by allowing multiple hardware transactions to run concurrently with one software transaction.

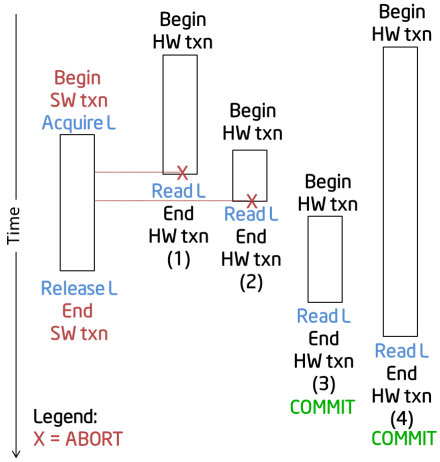



**Figure 2.** Lazy SGL (L-SGL).

Haswell RTM provides an abort status code that offers limited information about why a hardware transaction aborted. L-SGL makes it easier to collect diagnostic information about failed hardware transactions from this abort status code. When an E-SGL hardware transaction is about to start, it makes sense to wait until the SGL

is free. As a result, eager subscription rarely aborts hardware transactions explicitly at the time of subscription, so transactions are much more likely to be aborted automatically in-flight. Therefore, the abort status code will report this abort as a conflict. By contrast, L-SGL's lazy subscription mechanism makes it more likely that transactions will be aborted explicitly on subscription, allowing the programmer to obtain more detailed diagnostic information because, in this case, the abort status code can indicate precisely that the abort was caused by the lock.

L-SGL is similar to E-SGL in that it does not require read or write annotations, it permits transactions to be arbitrarily nested, but does not permit explicit transaction aborts in user code.

A software transaction waiting to acquire the SGL uses a combination of backoff and sleeping to reduce cache line contention. It starts by inserting an exponentially increasing number of null operations (NOPs) between successive lock attempts. When the number of NOPs reaches a threshold, $T$, the transaction calls the sleep function to release the processor for a brief duration before trying again. We found that sleeping right away is generally too slow for benchmarks where transactions are small and fast, but works well for larger and slower running transactions. Overall, we found that exponential waiting followed by sleeping works best across the range of benchmarks we considered.

Before a thread starts a hardware transaction, it reads the SGL to prefetch the lock into the cache. If no software transaction tries to acquire that lock, the lock is likely to be cached at commit time, which our experiments have observed to speed commit.

### 3.1 Correctness

STM designers often go to great efforts to ensure that all transactions see a consistent state, even after synchronization conflicts have occurred, a property called *opacity* [11]. The L-SGL design is simplified because hardware transactions do not need opacity. Instead, the L-SGL design relies on two guarantees. First, Haswell's hardware sandboxing mechanism ensures that any hardware transaction that raises an exception or enters an infinite loop because of an inconsistent state is aborted and rolled back without affecting any other transactions. Second, the L-SGL design ensures that no hardware transaction can commit while a software transaction is in progress. There is one exception, explained in the next section.

Fig. 3 illustrates why opacity is unnecessary: variables $X$ and $Y$ are linked by the invariant $Y = X + 1$. Now suppose a hardware transaction reads $X$ and $Y$ after a software transaction has incremented $X$, but before it has incremented $Y$, resulting in the inconsistent view $X = Y$. This hardware transaction will never commit, but it may encounter a segmentation fault when it evaluates $1/(Y - X)$. The Haswell hardware sandboxing mechanism will suppress the exception and roll back the transaction, ensuring that no other transaction is affected.

Figs. 4 and 5 outline possible orderings between hardware and software transactions. We order transactions by their commit time. Because software transactions cannot abort, any conflicting operation a software transaction executes after a hardware transaction has committed must be ordered after the hardware transaction. Moreover, because TSX provides no "escape actions" a hardware transaction cannot wait for a software transaction to commit.

In cases 1 (Fig. 4(a)) and 2 (Fig. 4(b)), the hardware transaction ends before the software transaction ends, and finds the lock taken

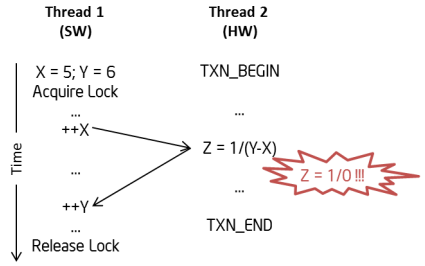



**Figure 3.** Inconsistent reads.

when it tries to commit. In these two cases, the hardware transaction must be serialized before the software transaction. If a software transaction performs an operation that conflicts with a concurrently executing hardware transaction while the hardware transaction is still in-flight, the hardware transaction is aborted by the Haswell HTM conflict detection mechanism. If, on the other hand, the conflicting operation is performed by the hardware transaction, the conflict would not be detected. If both transactions were permitted to commit, the value of the conflicting location would be incorrect because the hardware overwrote the software transaction's write (see Fig. 4). Here, we must abort the hardware transaction, because software transactions cannot be aborted. It does not matter when the hardware transaction is aborted, so it is sufficient to check for conflicts as the final step of the hardware transaction before it commits. In L-SGL, such conflicts are detected by inspecting the state of the lock.

In cases 3 (Fig. 4(c)) and 4 (Fig. 5), the hardware transaction begins its commit after the concurrent software transaction has committed. If the lock is free at the time of the hardware commit, then the hardware transaction can commit even though it might have overlapped one or more software transactions. Because the hardware transaction commits after any concurrently executing software transaction, it will be ordered after any such overlapping software transaction. Therefore the correct value for any conflicting location is the value written by the hardware transaction. If the last value written to a location that conflicts with the hardware transaction belonged to the software transaction, then the hardware transaction would have aborted, because Haswell's HTM conflict detection system would have identified such a conflict and aborted the hardware transaction. Moreover, a software transaction observes only old values until the hardware transaction commits, so the software reads are serialized before the hardware writes.

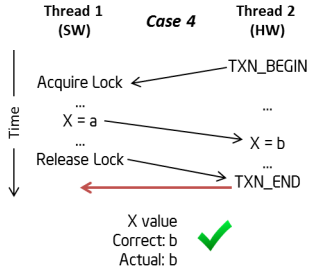
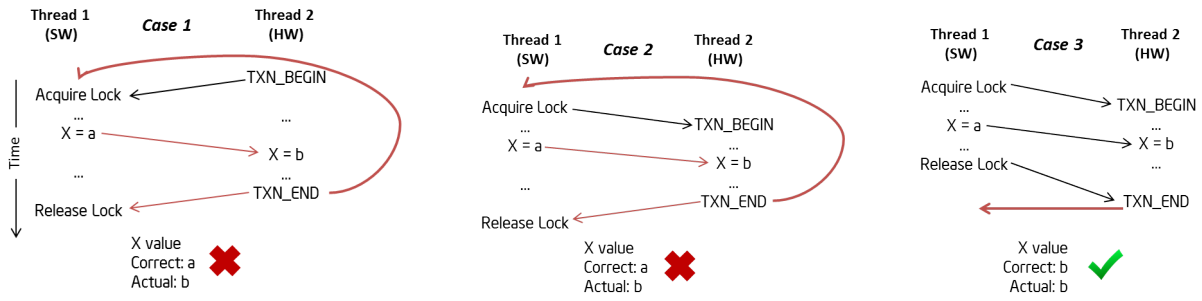



**Figure 5.** Correctness: Case 4.

### 3.2 Sandboxing

Hardware sandboxing prevents faults that occur inside a hardware transaction from propagating outside of the transaction. Spurious




(a) Case 1 (b) Case 2 (c) Case 3

**Figure 4.** Correctness: Cases 1-3. Arrows denote the "happens-before" relation.

writes and faults caused by reading inconsistent state from the SGL transaction are not visible to other threads. There is, however, one unlikely situation when inconsistent reads can cause a hardware transaction to commit prematurely. In principle, inconsistent reads could cause a spurious write to a location that is later used by the same transaction as the target of an indirect jump. If the target of the incorrect jump is is an xend (commit) instruction, or data that looks like one, then the hardware transaction might commit incorrectly, without checking the lock. Note that the inconsistent transaction cannot actually change the program code and insert spurious xend instructions, as the code area is protected and accessing it would cause the transaction to abort.

To address this hazard, lazy subscription must be performed before any indirect jump executed inside a hardware transaction that has written to memory. A read-only transaction, or one that is read-only before the indirect jump is not subject to this hazard. Moreover, if a transaction makes multiple indirect jumps, it is sufficient to check the lock only before the first jump, because the lock remains in the transaction's read set.

In the results presented in Section 4, we use L-SGL with early subscription on the first indirect jump that occurs after a shared memory write. We found that this restriction did not affect performance.

In general, this problem is similar to security concerns caused by buffer overflows. There is a trend towards compiler support to help with this issue, which might also be used to protect hardware transactions from incorrect premature commits. For example, the latest GCC supports security functionality to check vtable integrity. Moreover, for optimizations levels higher than -O2, GCC uses devirtualization and inlining for the most likely target in indirect jumps. A transactional compiler could use similar techniques to generate multiple likely targets and use the early lock check only in the unlikely case that none of the pre-established targets are chosen.

## 4. Experimental Evaluation

Our experimental evaluation was performed on an Intel Haswell processor (Core i7-4770) with RTM and HLE enabled, running at 3.40 GHz. The machine has a total of 8GB of RAM shared across four cores, each having a 32 KB L1 cache. For our experiments, hyper-threading was enabled, giving us a total of eight hardware threads. However, we notice that hyper-threading negatively impacts performance at 8 threads due to L1 cache sharing. In practice, this results in more hardware transactions being aborted because of overflow. To show this effect, we performed a simple experiment in which we measured the rate of aborts due to overflow for one, two, four and eight threads for all STAMP benchmarks. The rate of overflow for 1 thread is indicative of the percentage of transactions that cannot succeed in hardware because of cache size or associativity limitations. As we increase the number of threads, the rate of overflow decreases, as more and more transactions abort because of conflicts with other transactions. However, for 8 threads, the rate of overflow significantly increases, showing the negative effects of hyper-threading, as can be seen for the Vacation High benchmark in Fig. 6. Results were similar for all other STAMP benchmarks, except for the Labyrinth benchmark, where most of the aborts are caused by unsupported instructions; we omitted these graphs due to space constraints.

We used GCC 4.8.2 compiler with -O3 optimization enabled and gcc intrinsics [16]. We used the STAMP benchmarks [4] to compare L-SGL's speedup relative to a single-threaded sequential execution with software only approaches - a state-of-the-art STM (TL2) and a single global lock (spinlock) without any transactional execution (SGL) - and with a hardware only solution (Haswell HLE). For HLE, we used a single global spin lock prefixed with HLE-Acquire and HLE-Release instructions to suggest that the critical section should be executed speculatively. If speculation fails, the critical section is retried non-speculatively, according to a hardware policy. We also compared to the naïve SGL implementation with eager subscription (E-SGL). We ran all methods five times and presented the median of the results. Variance was generally low. We also compared L-SGL's rate of transactional success with that of HLE and E-SGL, by measuring the percentage of transactions executed non-speculatively for both methods.

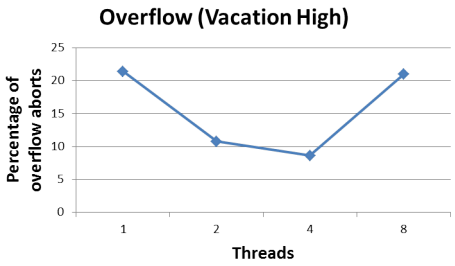
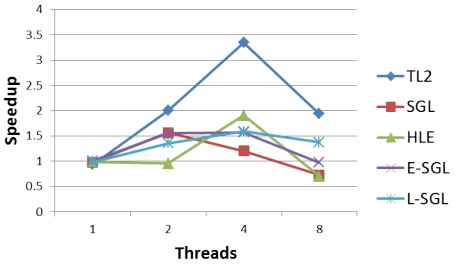
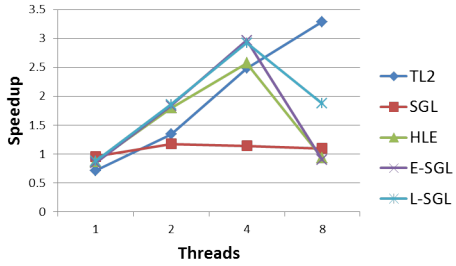


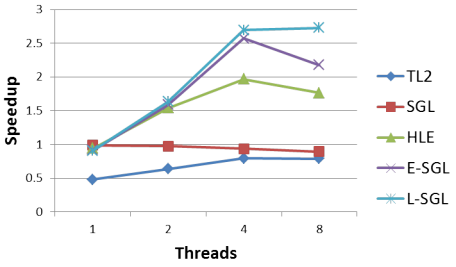

**Figure 6.** Example of overflow due to hyper-threading (vacation high benchmark).



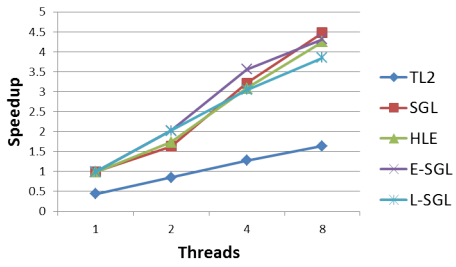

(a) Bayes.



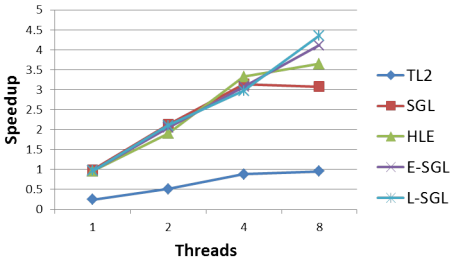

(b) Genome.



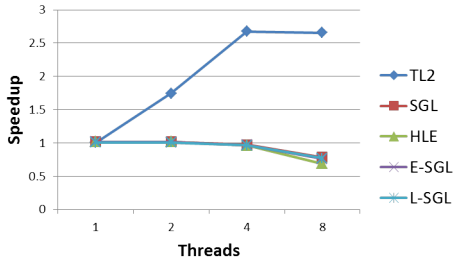

(c) Intruder.



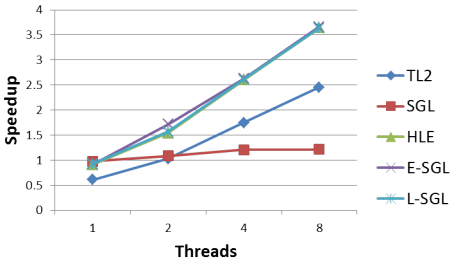

(d) Kmeans Low.



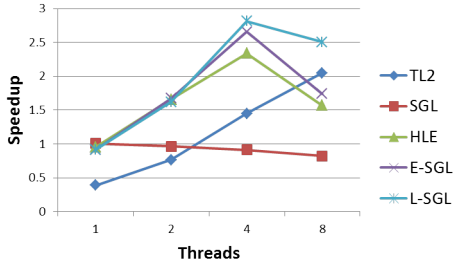

(e) Kmeans High.



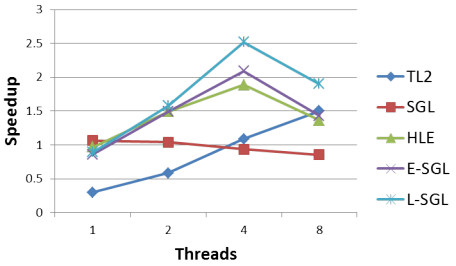

(f) Labyrinth.



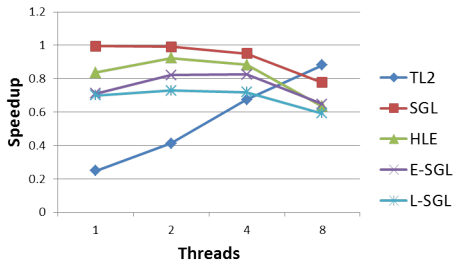

(g) Ssca2.




(h) Vacation Low.

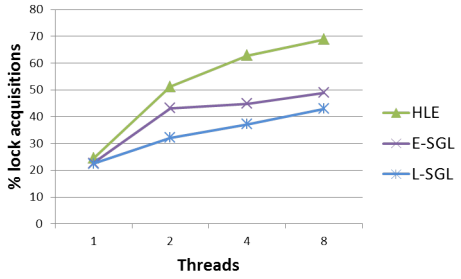



(i) Vacation High.

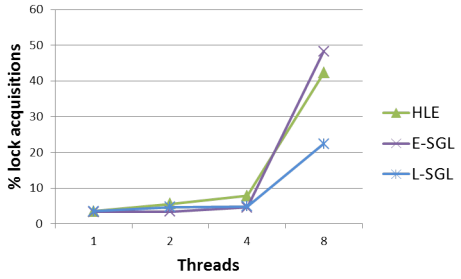



(j) Yada.

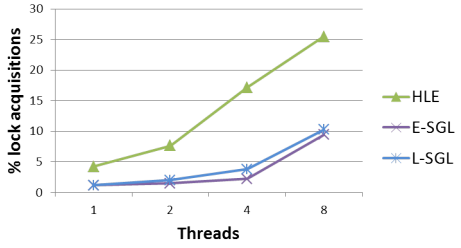
**Figure 7.** STAMP Throughput.



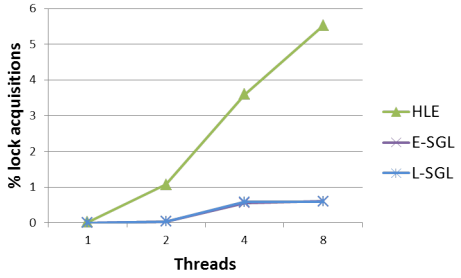

(a) Bayes.



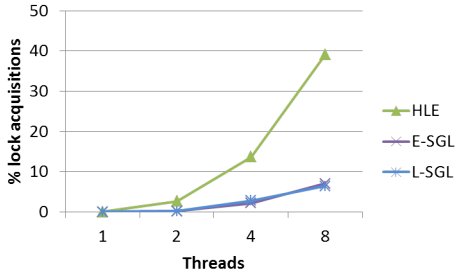

(b) Genome.



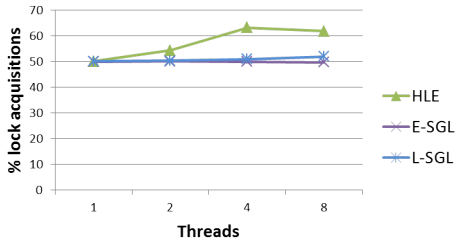

(c) Intruder.



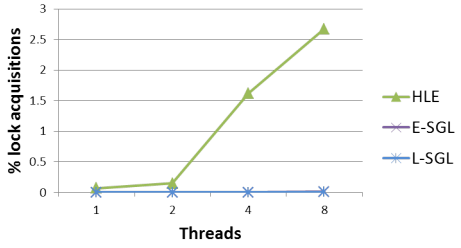

(d) Kmeans Low.



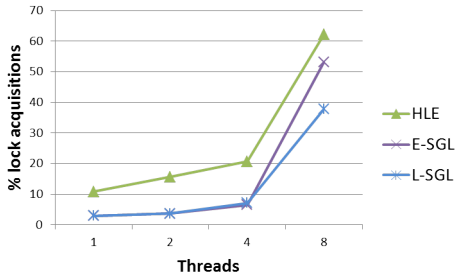

(e) Kmeans High.



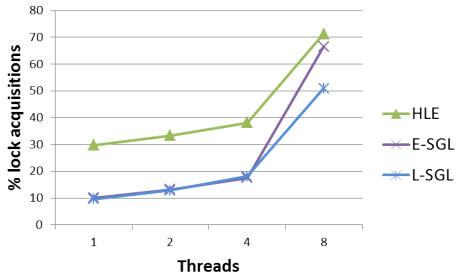

(f) Labyrinth.



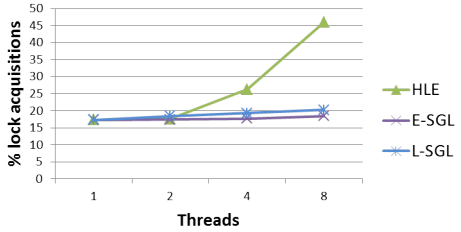

(g) Ssca2.




(h) Vacation Low.

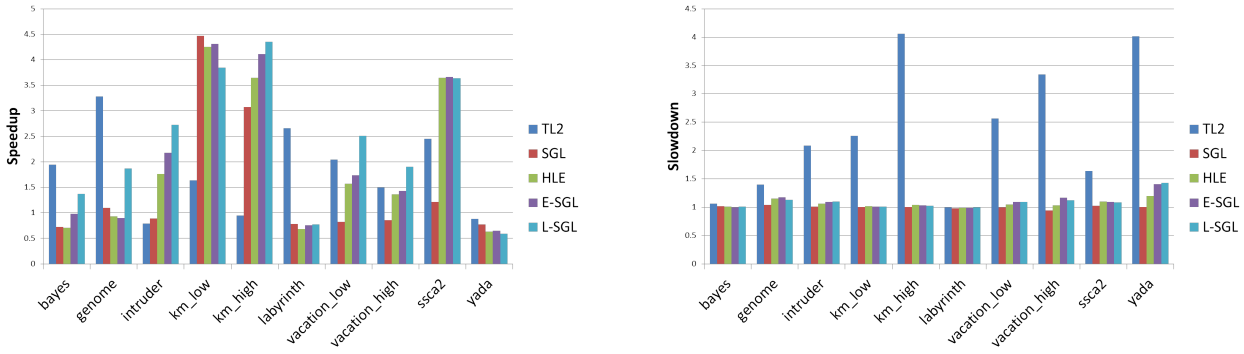



(i) Vacation High.




(j) Yada.

**Figure 8.** STAMP Percentage of Lock Acquisitions.




(a) Speedup for 8 threads

(b) Slowdown for 1 thread

**Figure 9.** Speedup for 8 threads and slowdown at 1 thread, compared to sequential.

### 4.1 Speedup relative to sequential execution

L-SGL performs best on benchmarks with medium sized transactions, such as Intruder 7(c), Vacation Low 7(h) and Vacation High 7(i), where it outperforms all prior methods. On the benchmarks with smaller transactions, such as Ssca2 7(g), Kmeans Low 7(d) and Kmeans High 7(e), L-SGL has good speedup compared to sequential execution, and outperforms TL2, which has too much overhead for these small transactions. However, L-SGL does not present a significant advantage compared to HLE on these benchmarks, because most transactions will quickly succeed in hardware, therefore making the differences between L-SGL and HLE less noticeable. For Kmeans Low 7(d), where there is little contention, SGL performs similar to L-SGL and HLE as well. However, when there is more contention, as is the case with Kmeans High 7(e), or when most transactions can succeed in hardware, in parallel, as in Ssca2 7(g), the performance of SGL quickly degrades.

Finally, for large transactions and those with unsupported instructions, as in Bayes 7(a), Labyrinth 7(f) and Yada 7(j), TL2 is more advantageous, because it can execute transactions in parallel, in software, without overflowing the cache. The effects of hyperthreading when running with 8 threads are even more pronounced on these benchmarks, because most transactions are large. We note that Labyrinth in particular is very suitable for STM systems because it uses very large transactions, whose initial memory accesses are all local. Therefore, these memory accesses do not contribute towards generating conflicts in an STM. Unfortunately, Haswell RTM does not have escape actions, therefore counting local accesses as transactional and overflowing the cache unnecessarily.

### 4.2 Percentage of lock acquisitions

We measured the percentage of lock acquisitions in L-SGL by inserting statistics in our code to measure the total number of transactions and the percentage executed non-speculatively. We measure the percentage of lock acquisitions in HLE using perf with support for TSX, a performance analysis tool for Linux.

We can notice in fig. 8 that L-SGL achieves a better rate of transactional execution than HLE on all STAMP benchmarks (its rate of lock acquisitions is lower than HLE's rate on all benchmarks). L-SGL uses lazy subscription, so the lock is read transactionally at the end of the critical section. In contrast, HLE subscribes to the lock address in the beginning of the critical section, suffering more aborts due to changes to the lock.

### 4.3 Single-threaded penalty

One of the biggest advantages of L-SGL is that it manages to improve performance for 4 and 8 threads without paying a big penalty for single threaded execution, as is the case with most STMs. For example, fig. 9(a) shows L-SGL's speedup relative to sequential for 8 threads and fig. 9(b) shows the slowdown for 1 thread. We can see that TL2 pays a huge penalty for single-threaded execution, while L-SGL execution is almost as good as sequential execution.

## 5. Fine-grained SGL

L-SGL allows multiple hardware transactions to execute concurrently with a software transaction as long as the software transaction commits first 4(c), 5. Unfortunately, hardware transactions that attempt to commit while a software transaction is in progress will abort 4(a), 4(b). This is the correct and expected behavior if there are conflicts between the hardware transactions and the software transaction, but otherwise these hardware transactions could successfully commit. Despite being an improvement over the simple single global lock algorithm, L-SGL does not enable the maximum amount of concurrency possible between multiple hardware transactions and a software transaction.

In this section, we describe another SGL fallback mechanism that performs finer grained conflict detection than E-SGL and L-SGL, based on Bloom filters (BF-SGL). BF-SGL increases the amount of concurrency offered by the hybrid transactional memory system in Fig. 4(a) and Fig. 4(b). In order to detect conflicts between the software transaction and hardware transactions, we add a Bloom filter for each thread. Each read and write is annotated to add the memory location to the Bloom filter. Hardware transactions consult the global lock before committing and, if they find it free, they can commit successfully. However, if the lock is taken, they can compare their Bloom filter with the software transaction's Bloom filter to determine if there are conflicts. The Bloom filter allows false positives, but not false negatives. Therefore, it could detect a conflict despite the transactions not having any conflicts, but it will never report zero conflicts if the transactions accessed the same memory. So the hardware transactions can commit successfully even if the lock is taken as long as the Bloom filters do not report

conflicts. L-SGL represents a particular case of BF-SGL. Specifically, L-SGL can be obtained from BF-SGL if the Bloom filter set intersection operation between the hardware transaction trying to commit and the currently executing software transaction always returns that there exists at least one conflict.

### 5.1 Use Cases

Using BF-SGL, many small hardware transactions that access disjoint memory locations and concurrently executing large software transactions can commit. The same is not true for any other system that we are aware of. This is because we provide precise conflict detection using the Bloom filters for the HW and SW transactions to track memory accesses. Consider, for example, an array representing an open addressing hash-table. Threads can perform lookup(x) operations and insert(x) operations in this hash-table. Once a threshold of occupancy is achieved, a thread decides to double the size of the hash-table by allocating a new array and rehashing elements from the old array to the new array. Lookup and insert are short transactions and can succeed in hardware most of the time. Rehashing is always executed as a software transaction, so the thread needs to acquire the single global lock.

Using L-SGL, no lookup and insert operations can succeed during rehashing. However, using BF-SGL with precise conflict detection between the software transaction and the concurrent hardware transactions, lookup operations executed as hardware transactions can commit using data from the old array while the rehashing to the new array is taking place. Moreover, insert operations executed as hardware transactions at the end of the old array, in the part that has not been rehashed yet, can also commit during rehashing. Therefore, BF-SGL improves throughput by allowing small hardware transactions to commit concurrently with long executing software transactions.

### 5.2 Performance and Practicality

Adding the software Bloom filter to hardware transactions adds some overhead compared to simple hardware transactions. However, the Bloom filter adds the benefit of being able to commit hardware transactions even when a software transaction is executing as long as there are no real conflicts or false conflicts caused by the Bloom filter. An efficient Bloom filter implementation allows insertion and set intersection in O(1) time, minimizing the overhead.

In addition, reading these two locations in the hardware transaction only adds two additional cache lines to the read set of the transaction. This can be optimized so that a bit of the Bloom filter is used to indicate whether the lock is taken or not and the rest is used as a Bloom filter. Therefore, the lock location can serve both purposes, reducing the read set size of the hardware transaction to just one additional location. The transaction's own Bloom filters add additional cache lines to the write set, but this could be as low as only one cache line, depending on the Bloom filter size.

Hardware transactions read the software Bloom filter only right before committing, narrowing the window when hardware transactions could be aborted by software transactions. Unfortunately, the software transaction needs to modify its Bloom filters for every read and write, causing many spurious aborts for the hardware transactions. We found that this behavior significantly affects the performance of BF-SGL, so we did not include results for this system. However, we note that this is a strong motivation why escape actions should be included with any HTM. If we had escape actions, the Bloom filters could be read non-transactionally at the end of the hardware transaction, avoiding the spurious aborts caused by the software transaction updating its Bloom filters. Correctness would still be maintained because any conflicting read or write performed by the software will still abort the hardware transaction. We believe this support will be available in the future, making the bloom filter based conflict detection a viable option. For example, IBM's Power ISA suspended mode [3] provides the necessary functionality.

## 6. Hardware Optimizations

***Lazy Hardware Lock Elision (LHLE).*** Haswell's HLE works by eliding locks prefixed with HLE-Acquire and executing the critical sections as hardware transactions. If the speculation fails for any reason, the lock is acquired and the critical section is re-executed non-speculatively in software. HLE is similar to E-SGL: hardware transactions need to subscribe to the lock in the beginning of their execution to ensure correctness. However, we have shown that L-SGL, implemented in software, outperforms the hardware only HLE. Therefore, we speculate that Lazy Hardware Lock Elision (LHLE), where the lock is added to the read set at the end of the critical section, would perform better than HLE. Similar to HLE, LHLE enables multiple speculative critical sections to execute in parallel if there are no conflicts detected at run-time and it simplifies programming by enabling more parallelism for coarse-grained critical sections. In contrast to HLE, LHLE supports parallelism between one non-speculative critical section and multiple speculative critical sections. Moreover, LHLE is designed to be implemented entirely in hardware, so the sandboxing issues described in Section 3.2 do not arise, as the hardware can ensure that the subscription to the lock occurs whenever the xend instruction is invoked.

***Bloom Filter Hardware Lock Elision.*** As described in Section 5, BF-SGL can improve the granularity of conflict detection with an SGL, but causes spurious aborts because the SGL transaction's Bloom filters become part of the read set of the hardware transactions. This could be avoided if the HTM allowed escape actions. In that case, the Bloom filters would be read non-transactionally to detect conflicts. Alternatively, if the Bloom filters were handled by the hardware instead of the software, they could avoid the tracking mechanism of HTM and avoid the unnecessary aborts. Haswell HLE could be extended with Bloom filters for the hardware transaction, as well as for the SGL transaction. With this design, conflict detection would be realized at a finer-grained level than it is currently done.

## 7. Conclusions

The naïve SGL fallback's simplicity makes it an appealing alternative to more complicated, even if better-performing, HyTM schemes. In this paper, we introduced novel SGL methods that improve the performance of the simple SGL fallback, while maintaining its simplicity. First, we described L-SGL, a simple SGL-based fallback for HTM that uses lazy subscription to allow hardware-software transaction concurrency. L-SGL improves performance on current machines by up to 4X compared to state-of-the-art software and hardware solutions.

In addition, L-SGL has some appealing properties. For example, it does not require read and write annotations, making it suitable for implementation in a real system, either in the compiler or even in hardware. Our L-SGL software implementation improves performance over native Haswell lock elision by almost a factor

of 2, and reduces the rate of lock acquisitions by up to 35%. We conjecture this difference would be even higher if L-SGL were implemented in hardware.

We also described BF-SGL, an alternative SGL fallback mechanism with more accurate conflict detection. Our BF-SGL results, perhaps counter-intuitively, show that adding a mechanism to support better conflict detection, such as Bloom filters, hinders performance by increasing the abort rate. If the HTM were to support escape actions, allowing precise conflict detection to be performed outside of transactional tracking, we speculate that this comparison would change in favor of BF-SGL. Finally, we showed how to use these ideas to improve future HTMs with minimal microarchitectural changes.

## Acknowledgments

We thank Justin Gottschlich, Konrad Lai, Andi Kleen and the anonymous reviewers for their useful feedback and suggestions.

## References

[1] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha. Transactional programming in a multi-core environment. In K. A. Yelick and J. M. Mellor-Crummey, editors, *PPoPP*, page 272. ACM, 2007.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. volume 26, pages 59–69. IEEE Computer Society Press, Los Alamitos, CA, USA, 2006.

[3] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 225–236, New York, NY, USA, 2013. ACM.

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

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

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

[7] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 336–346, New York, NY, USA, 2006. ACM.

[8] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIV, pages 157–168, New York, NY, USA, 2009. ACM.

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

[10] J. E. Gottschlich, M. Vachharajani, and J. G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, April 2010.

[11] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.

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

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*. May 1993.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, Inc., 2008.

[15] Intel Corporation. Hardware lock elision in Haswell. Retrieved from `http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-win/GUID-A462FBC8-37F2-490F-A68B-2FFA8010DEBC.htm`.

[16] Intel Corporation. Transactional Synchronization in Haswell. Retrieved from `http:/http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`, 8 September 2012.

[17] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[18] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, aug 2007.

[19] V. J. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 227–236, New York, NY, USA, 2008. ACM.

[20] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 221–228, New York, NY, USA, 2007. ACM.

[21] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: the importance of nonspeculative operations. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 53–64, New York, NY, USA, 2011. ACM.

[22] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPOPP*. ACM SIGPLAN 2006, Mar. 2006.

[23] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[24] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.