HAL
open science

Amalgamated Lock-Elision

Yehuda Afek, Alexander Matveev, Oscar R. Moll, Nir Shavit

To cite this version:

Yehuda Afek, Alexander Matveev, Oscar R. Moll, Nir Shavit. Amalgamated Lock-Elision. DISC 2015, Toshimitsu Masuzawa; Koichi Wada, Oct 2015, Tokyo, Japan. 10.1007/978-3-662-48653-5_21 . hal-01206577

HAL Id: hal-01206577

https://hal.science/hal-01206577

Submitted on 29 Sep 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Amalgamated Lock-Elision

Yehuda Afek[1], Alexander Matveev[2], Oscar R. Moll[3], and Nir Shavit[4]

[1] Tel-Aviv University, afek@post.tau.ac.il
[2] MIT, amatveev@csail.mit.edu
[3] MIT, orm@mit.edu
[4] MIT and Tel-Aviv University, shanir@csail.mit.edu

Abstract. Hardware lock-elision (HLE) introduces concurrency into legacy lock-based code by optimistically executing critical sections in a *fast-path* as hardware transactions. Its main limitation is that in case of repeated aborts, it reverts to a *fallback-path* that acquires a serial lock. This fallback-path lacks hardware-software concurrency, because all fast-path hardware transactions abort and wait for the completion of the fallback. Software lock elision has no such limitation, but the overheads incurred are simply too high.

We propose *amalgamated lock-elision* (ALE), a novel lock-elision algorithm that provides hardware-software concurrency and efficiency: the fallback-path executes concurrently with fast-path hardware transactions, while the common-path fast-path reads incur no overheads and proceed without any instrumentation. The key idea in ALE is to use a sequence of fine-grained locks in the fallback-path to detect conflicts with the fast-path, and at the same time reduce the costs of these locks by executing the fallback-path as a series segments, where each segment is a dynamic length short hardware transaction.

We implemented ALE into GCC and tested the new system on Intel Haswell 16-way chip that provides hardware transactions. We benchmarked linked-lists, hash-tables and red-black trees, as well as converting KyotoCacheDB to use ALE in GCC, and all show that ALE significantly outperforms HLE.

Keywords: Multicore, Hardware Lock Elision, Hardware Transactional Memory, Algorithms

1 Introduction

Hardware lock-elision (HLE) [30] introduces concurrency into lock-based critical sections by executing these sections in a *fast-path* as hardware transactions. However, hardware transactions are *best-effort* in current Intel Haswell [31] and IBM Power8 [8] processors, which means that they have no progress guarantee: a hardware transaction may always fail due to a hardware-related reason such as an L1 cache capacity limitation, an unsupported instruction, or a page protection or scheduler interrupt; in all such cases it may never commit [17]. Therefore, to ensure progress in HLE, a critical section that repeatedly fails to commit in the hardware fast-path, reverts to execute in a fallback-path that acquires the original serial lock. This fallback-path is expensive because it aborts all current fast-path hardware transactions and executes serially.

Recent work by Afek et al. [32] and Calciu et al. [24] introduced the *lock-removal* (or lazy subscription) lock elision scheme. Lock-removal sacrifices safety guarantees in favor of limited concurrency: the fast-path can execute concurrently with the fallback-path, however, the fast-path cannot commit as long as there is a fallback in process, and can observe inconsistent memory states. These inconsistent states can lead to executing illegal instructions or to memory corruption. It was claimed that HTM sandboxing significantly minimizes chances of unsafe executions, since any inconsistent hardware transaction should simply abort itself, and therefore, one can provide an efficient software-based compiler and the necessary runtime support to detect and handle the unsafe cases that "escape" the HTM sandboxing mechanism. Unfortunately, recent work by Dice et al. [12] shows that this is not the case: there are new cases of unsafe executions, ones not identified in the original lock-removal HLE papers, and require complex compiler and runtime support that would slow lock-removal HLE to a point that eliminates the advantages of using it in the first place. Instead, Dice et al. [12] propose new hardware extensions that can provide a fully safe HTM sandboxing capability.

Roy, Hand, and Harris [2] proposed an all software implementation of HLE in which transactions are executed speculatively in software, and when they fail, or if they cannot be executed due to system calls, the system defaults to the original lock. Their system instruments all object accesses, both memory reads and writes, and employs a special kernel-based thread signaling mechanism. This software based system provides better concurrency than HLE but introduces *software-software concurrency* that complicates the required compiler and runtime support and results in a slow scheme. As an example of a possible software concurrency complication, consider the execution shown in Figure 1(B), where thread P removes a node from a linked-list, and thread Q concurrently reads the removed node. In this case, thread P also modifies the removed node outside the critical section, which results in a divide by zero exception in thread Q. In the original code that uses a lock instead of atomic blocks (software transactions), this erroneous behavior is impossible. This problem is also known as privatization-safety [25, 28, 1, 2, 10]. Fixing this problem in software is expensive, and Attiya et al. [6] show that there is no way to avoid these costs. Afek, Matveev and Shavit [4] proposed PLE, an all-software version of HLE for read-write locks that uses a fully pessimistic STM, but still has software-software concurrency that results in using the expensive quiescence mechanism [10, 21, 22]. A recent paper by Dice et al. [13] proposes to integrate both hardware and software into an adaptive scheme, but has a software mode that has software-software concurrency as well, and proposes manual code modifications to avoid software costs.

In this paper we propose *amalgamated lock-elision* (ALE), a novel lock-elision algorithm that provides both hardware-software concurrency and efficiency: the fallback-path executes concurrently with fast-path hardware transactions, while the common-path fast-path reads incur no overheads and proceed without any instrumentation. The problem is that instrumenting all reads and writes in hardware simply makes it as slow as the software path, so why use hardware in the first place. The key idea in ALE is to use a sequence of fine-grained locks in the fallback-path to detect conflicts with the fast-path, and at the same time, reduce the costs of these locks by executing the fallback-path as a series segments, where each segment is a dynamic length short hardware transac-

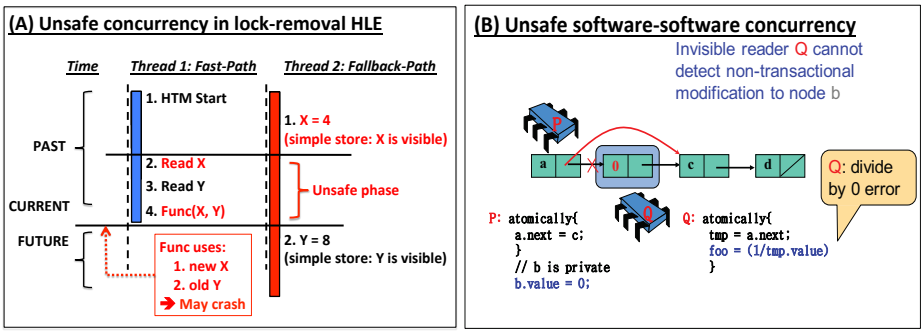

Fig. 1. (A) An example of unsafe hardware-software concurrency in lock-removal HLE. (B) An example of unsafe privatization due to software-software concurrency.

tion. Also, ALE forbids software-software concurrency, so that there is no need for complex and expensive software support that it would otherwise require. Instead, the focus in ALE is on making hardware-software concurrency efficient.

The new ALE protocol combines several ideas. First, it uses a "mixed" fallback-path that uses both software and hardware, in the style of [26, 27], to preserve full safety guarantees while the fallback-path executes concurrently with fast-path hardware transactions. To understand the problem in doing so, Figure 1(A) presents a simple unsafe scenario of lock-removal HLE [32, 24], that may occur when the fallback-path of Thread 2 executes concurrently with a fast-path hardware transaction of Thread 1. As can be seen in the figure, first the fallback updates X, and then the hardware transaction reads both X and Y. But, since the fallback has not yet updated Y, the hardware transaction has read a new X and an old Y relative to the fallback. As a result, at this point in time, the hardware transaction executes on an inconsistent memory state and may perform random operations that may corrupt memory or even crash the system [12]. In ALE, the mixed fallback-path defers the writes to the commit, and executes all of the writes in a "one shot" short hardware transaction, so that intermediate states (some mix of old and new values) are never visible.

ALE combines the mixed fallback-path with fine-grained locks to provide concurrency between fast-path hardware transactions and the fallback-path. More specifically, a lock ownership array, in the style of [20, 11, 29, 14], coordinates the reads of the fallback-path with the writes of the fast-path. Figure 2(A) depicts this coordination, where Thread 1 executes a fast-path hardware transaction and Thread 2 a fallback-path. On start, both read X and Y, and the fallback-path also locks the associated locks of X and Y. Then, the fast-path writes to X, but before actually committing it first verifies that X is unlocked. In this way, the fast-path cannot overwrite the fallback-path reads. In Figure 2(A), X is locked, so the fast-path of Thread 1 detects this conflict and aborts. However, if there were no real conflict, the fast-path would be able to commit concurrently.

Figure 2(A) also shows the use of a write-buffer in the fallback-path. The write-buffer is necessary to delay the actual writes to the commit-phase, where the ALE

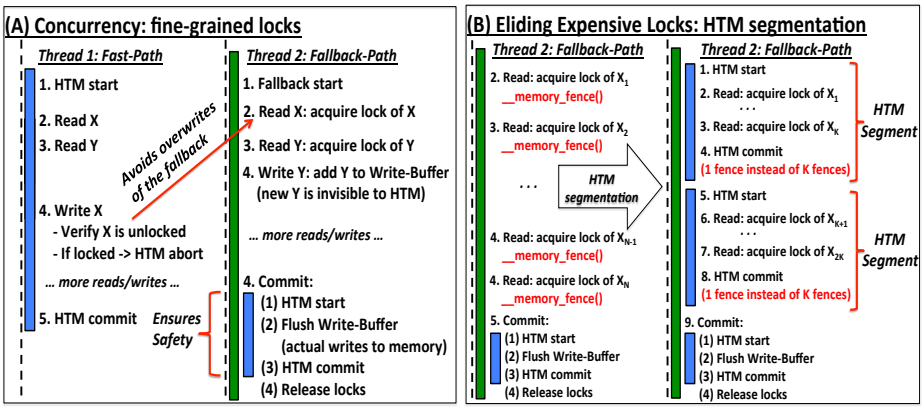

Fig. 2. (A) The mixed fallback-path of ALE uses fine-grained locks for conflict detection, and a write-buffer that defers actual writes to the commit, where all writes execute in a short "one-shot" hardware transaction. (B) The fallback-path executes as a series of segments, where each segment is a short hardware transaction that reduces the cost of lock barriers it includes to one barrier (the HTM commit).

protocol will execute all of the writes as a "one shot" short hardware transaction. In the unlikely case this short hardware transaction fails, ALE acquires the critical section lock, that aborts all hardware transactions, and executes the writeback as is (in software). Notice that this aspect of ALE is similar to HLE, however, hardware transactions in ALE abort and block only for the short duration of the writeback and not for the whole fallback-path execution.

The downside of having a lock for each read of the fallback-path is the expensive memory barriers: a lock acquire must execute a memory barrier to become visible immediately, which forces the processor to drain the store buffer on each lock acquire, before proceeding to the next instruction. To overcome this overhead, ALE elides most of the memory barriers of lock acquires, by executing the fallback-path as a series of segments, where each segment executes as a short hardware transaction. A lock acquire that executes inside a hardware transaction does not need to become visible immediately, and therefore, does not require a memory barrier. As a result, multiple lock acquires that execute in a hardware segment, involve using only one barrier (instead of many) that executes as part of the hardware commit. Figure 2(B) depicts this idea: on the left, the fallback-path executes without segmentation, which involves executing a memory barrier for each read, while on the right, the hardware segmentation allows to reduce the number of barriers from the number of reads to the number of segments.

What if a hardware segment fails to commit? For this purpose, ALE implements a dynamic segmentation [5] policy that adjusts the length of each segment based on hardware aborts. In particular, the protocol counts the number of reads and writes that execute in the hardware segment, and when this count reaches a predefined limit, it commits the hardware segment and starts a new one. The predefined limit is dynamic: it gets reduced on excessive aborts, and gets increased on successive commits. In the

extreme case, when a segment cannot commit (unsupported instruction or some interrupt), it reverts to execute in the standard software mode, where each read involves a memory barrier. It is important to notice that subsequent segments may still commit in the hardware, so the fallback is local for the specific segments that repeatedly fail.

We implemented ALE in GCC using the recent transactional memory support [3] and tested it on an Intel Haswell 16-way chip. We executed micro-benchmarks including linked-lists, hash-tables and red-black trees, which all show that ALE is significantly more performant than HLE due to the additional concurrency it provides. In addition, we tested ALE by converting KyotoCacheDB [18], a commercially used database management library, from read-write locks to the GCC based ALE. Our results are encouraging and show that the ALE implementation is two times faster than an HLE one.

2 Amalgamated Lock-Elision

2.1 Algorithm Overview

The key challenge in the design of ALE is efficiency: it is hard to provide (1) fallback-path that can execute concurrently with fast-path hardware transactions and preserve (2) full safety guarantees. This is why researchers propose to sacrifice safety [32, 24] or introduce new hardware extensions [12]. In contrast, ALE provides these properties on commodity multicore architectures of Intel and IBM.

In a nutshell, the ALE protocol works as follows. The fallback mechanism of ALE is similar to HLE: a critical section first starts as a fast-path hardware transaction, and only when it fails multiple times to commit in the hardware, it reverts to execute in the fallback-path.

Safe concurrency. ALE mixes fine-grained locks and short hardware transactions in the fallback-path to detect all possible conflicts between fast-path hardware transactions and the fallback-path. First, the fallback-path locks each read location at encounter-time, while fast-path hardware transactions verify the locks of locations they want to write at commit-time. In this way, the fast-path writes detect fine-grained conflicts with the fallback-path reads. Second, the fallback-path writes are buffered and delayed to the commit-phase, where they execute in a "one shot" short hardware transaction. As a result, the hardware detects all conflicts involving fallback-path writes. Put together, both techniques ensure that all conflicts between fast-path hardware transactions and the fallback-path are detected, which allows safe concurrency between the two. Notice, that in the case when there is a conflict, the one that aborts is the fast-path and not the fallback-path. Finally, ALE executes fallbacks one at a time, so there are no possible fallback-to-fallback conflicts.

Efficiency. A lock acquire involves executing an expensive memory barrier, and therefore, a fallback-path that involves many reads, will result in many lock acquires that introduce an unacceptable performance penalty. To reduce locking costs, the fallback-path executes as a series of segments, where each segment executes as a short hardware transaction. All lock acquires that execute within a single hardware segment become visible atomically when the hardware segment commits, and therefore have no need

to execute individual barriers. This allows multiple lock acquires to share the cost of a single barrier (of the hardware commit). Because hardware transactions may abort, ALE splits the fallback-path dynamically: it adapts each segment length to the specific abort behavior of the code, and falls back to execute the standard locking barriers of the segment, when the segment repeatedly fails to commit.

Naturally, the benefit of ALE depends on the success ratio of the short hardware transactions that execute in the fallback-path. In particular, a high success ratio of the fallback-path writeback is necessary to avoid excessive fast-path aborts. Our empirical results show that this is usually the case, because most operations follow the 80:20 rule (80% reads and 20% writes [23]). In general, other read-write distributions are possible, but we believe that our benchmarks that present a set of popular data-structures and a real-world application are encouraging. In addition, besides the writeback, the fallback-path segments may repeatedly fail to commit in hardware. For example, a segment will always fail when it will try to execute an unsupported instruction or encounter a page fault interrupt. In our experiments, most of the segments succeed to adapt to the specific code behaviors, and only in rare cases, segments must revert to software and execute one barrier per lock.

2.2 Algorithm Details

Algorithm 1 and Algorithm 2 present the pseudo-code for ALE fast-path and fallback-path: each critical section first tries to execute in the fast-path as a hardware transaction, and when it repeatedly fails to commit it reverts to execute in the fallback-path. The pseudo-code assumes an elision process for a critical section that is protected by a lock named section-lock, and presents a simplified version of the code that omits the code that handles nested locks and hardware segmentation (similar to [5]).

Global Structures The ALE protocol is based on the following global structures:

- locks-array : An ownership array, in the style of [11, 29], that uses a hash function to assign a 64bit lock for each memory location (all initially 0). In our implementation we allocate an array with 2^{19} locks and use a "striping" hash function that maps each consecutive 2^8 bytes of memory to the same lock.
- fallback-lock : A 64bit lock that is also used as a counter by the fallback-path to lock locations. A zero value represents that there is no fallback executing (initially 0).
- acquire-counter/release-counter : 64bit counters that the fallback-paths use to execute one at a time (both initially 1).

In ALE all locks and counters only increase, except for the fallback-lock that alternates between 0 and ever increasing number. This is why ALE uses 64bit counters to avoid overflows.

Fast-Path Algorithm 1 presents the pseudo-code for the fast-path. On start, it resets a local lock-id-log to an empty set, and then initiates a hardware transaction (lines 2 - 3).

Algorithm 1 ALE: fast-path

```
1:  function FAST_PATH_START(ctx)
2:      ctx.lock-id-log ← ∅
3:      while HTM_START() = htm-failed do
            ▷ Outside HTM
4:          ... some fast-path retry policy ...
5:          if no retry then
6:              ... switch to fallback-path ...
            ▷ Inside HTM
7:      if section-lock ≠ 0 then
8:          HTM_ABORT()
9:
10: function FAST_PATH_READ(ctx, addr)
11:     return load(addr)                ▷ Direct read
12:
13: function FAST_PATH_WRITE(ctx, addr, v)
14:     lock-id ← HASH(addr)
15:     ctx.lock-id-log ∪ = {lock-id}
16:     store(addr, v)                   ▷ Direct write
17:
18: function FAST_PATH_COMMIT(ctx)
19:     if ctx.lock-id-log = ∅ then
20:         HTM_COMMIT()                 ▷ read-only
21:         return
22:     if fallback-lock = 0 then
23:         HTM_COMMIT()                 ▷ no fallback
24:         return
25:     for id ∈ ctx.lock-id-log do
26:         if lock-array[id] = fallback-lock then
27:             HTM_ABORT()              ▷ conflict found
28:     HTM_COMMIT()                     ▷ no conflicts
```

Then, inside the hardware, it first puts the section-lock into hardware monitoring, by verifying that this lock is free (lines 7 - 8). This step provides the fallback-path with an ability to abort all hardware transactions (for the case when the fallback writeback short hardware transaction fails). Next, during the execution, the reads proceed directly without any instrumentation (line 11), while the lock ids of writes are logged to the lock-id-log (lines 14 - 15). On commit, if the fast-path transaction has been read-only or detects that there is no fallback (lines 19 - 24), then it can simply commit. Otherwise, the fast-path traverses each logged lock-id and verifies that it is free (lines 25 - 27). If some lock is not free, then there is a conflict with a fallback read, and the fast-path aborts, else the fast-path commits safely and concurrently.

Fallback-Path Algorithm 2 presents the pseudo-code for the fallback-path. On start, it resets a local write-log to an empty set, and then acquires the fallback-lock (lines 2 - 3). Next, it initiates the process of hardware segmentation (line 4) that elides the expensive per lock barriers. During the execution, both read and write first execute the segmentation checkpoint function (lines 6 , 17), that controls the dynamic split of hardware segments (increments a local counter of reads/writes and splits the segment when it reaches a limit). Then, on write, it simply buffers the write into the write-log (line 18), and on read, it first checks if the read location is in the write-log (lines 7 - 8). If this is the case, then it returns the value from the write-log, else it proceeds to locking the location and reading its value from the memory (lines 9 - 15).

As can be seen in the read procedure, to lock a read location, the ALE writes the fallback-lock into the lock, and only executes the actual barrier of the lock if the current segment reverted to the software. Notice that only a single fallback can execute at a time, and therefore, the fallback-lock is actually specific to the fallback that currently executes. This allows a fast-path hardware transaction to identify that a lock in the locks-array is taken, by checking that the lock value equals to the current value of the

Algorithm 2 ALE: fallback-path

```
1: function FALLBACK_START(ctx)
2:     ctx.write-log ← ∅
3:     ACQUIRE_FALLBACK_LOCK(ctx)
4:     HTM_SEGMENT_START()

5: function FALLBACK_READ(ctx, addr)
6:     HTM_SEGMENT_CHECKPOINT()
7:     if (addr, val) ∈ ctx.write-log then
8:         return val
9:     lock-id ← HASH(addr)
10:    lock-array[lock-id] ← fallback-lock
11:    if HTM_ACTIVE() then
12:        return load(addr)        ▷ Elide barrier
13:    else
14:        MEMORY_BARRIER()         ▷ no HTM
15:        return load(addr)

16: function FALLBACK_WRITE(ctx, addr, v)
17:    HTM_SEGMENT_CHECKPOINT()
18:    ctx.write-log ∪ = {addr, v}

19: function FALLBACK_COMMIT(ctx)
20:    HTM_SEGMENT_COMMIT()
21:    WRITE_BACK(ctx)
22:    RELEASE_ALL_LOCKS(ctx)

23: function WRITE_BACK(ctx)
24:    while HTM_START() = htm-failed do
25:        ... some write-back retry policy ...
26:        if no retry then
27:            section-lock ← 1     ▷ aborts HTM
28:            FLUSH(ctx.write-log)
29:            section-lock ← 0  ▷ resumes HTM
30:            return
31:    FLUSH(ctx.write-log)
32:    HTM_COMMIT()

33: function ACQUIRE_FALLBACK_LOCK(ctx)
34:    turn ← FETCH&ADD(acquire-counter)
35:    while release-counter ≠ turn do
36:        spin-wait            ▷ wait for my turn
37:    fallback-lock ← turn
38:    MEMORY_BARRIER()

39: function RELEASE_ALL_LOCKS(ctx)
40:    fallback-lock ← 0 ▷ releases all locks
41:    FETCH&ADD(release-counter)
```

fallback-lock. As a result, the fallback-path can release all locks at once, by a single write that resets the fallback-lock. The next fallback-path will use a subsequent value of the acquire-counter, and therefore, all previous locks will not be seen as taken (lines 34 - 41).

The segmentation process of ALE is dynamic: it adjusts the length of each segment based on hardware aborts it encounters. More specifically, it counts the number of reads and writes in each segment checkpoint call, and when this count reaches a predefined limit, then it initiates a split procedure: commits the current segment and starts a new one. On start, the predefined limit is set to a large value (100 shared accesses), and during the execution gets reduced by 1 on a hardware abort, and gets increased by 1 on 4 successive hardware commits. This simple algorithm could be tuned and made more adaptive.

The success ratio of segmentation also depends on the implementation of the write-set buffer lookup function. If the lookup traverses the whole buffer on each read, then potentially it may introduce excessive HTM capacity aborts into the segments. To avoid this negative effect, ALE implements the write-set buffer as a hash table with 64 buckets, and uses a bloom filter [11, 7] to minimize lookups.

HTM Retry Policy Our empirical evaluation shows that the HTM retry policy is performance critical (also shown in [16, 32]). We implement a simple policy that we found to perform well. When a fast-path hardware transaction fails, ALE checks the abort code, and if the *IS_RETRY* flag is set, then it retries the fast-path. Else, it reverts to the fallback-path. The limit of retries is set to 10. The short hardware transaction of the fallback-path gets retried in a similar way, while the fallback-path hardware segments retry based on the adaptive segmentation [5].

3 Performance Evaluation

We benchmarked using an Intel Core i7-5960X Haswell processor with 8-cores and support for HyperThreading of two hardware threads per core. This chip provides support for hardware transactions that can fit into the capacity of the L1 cache. It is important to notice that the HyperThreading reduces the L1 cache capacity for HTM by a factor of 2, since it executes two hardware threads on the same core (same L1 cache). As a result, in some benchmarks there is a significant penalty above the limit of 8 threads, where the HyperThreading executes and generates an increased amount of HTM capacity aborts.

The operating system is a Debian OS 3.14 x86_64 with GCC 4.8. We added the new ALE scheme into GCC 4.8 that provides compiler and runtime support for instrumenting shared reads and writes and generated two execution paths (the fast-path and the fallback-path), as part of the GCC TM draft specification for C++ [3]. Our results show that the malloc/free library provided with the system is not scalable and imposes significant overheads and false aborts on the HTM mechanism. As a result, we used the scalable *tc-malloc* [19] memory allocator, which maintains local per thread pools.

We compared the following lock elision schemes:

1. *HLE:* This is the state-of-the-art hardware lock elision scheme of Rajwar et al. [30], in which we also implement the advanced fallback mechanism (as described in 2.2 and also noted in [16]). This scheme provides full safety guarantees, but has no concurrency between fast-path hardware transactions and the fallback-path. More specifically, a lock-based critical section starts in the fast-path as a hardware transaction, and then immediately verifies that the lock of this section is free. In this way, when the fast-path repeatedly fails to commit, it reverts to execute in serial mode, in which it acquires the section lock that triggers an abort of all fast-path hardware transactions.
2. *HLE-SCM:* This scheme combines HLE with software-assisted contention management [32]: it introduces an auxiliary lock to serialize fast-path hardware transactions that repeatedly abort due to conflicts. In this way, it reduces unnecessary hardware conflicts under high contention, and increases the success probability of the fast-path.
3. *Unsafe-LR:* The unsafe lock-removal (lazy subscription) lock elision scheme [32, 24] that provides improved concurrency: the fallback-path can proceed concurrently with hardware transactions, however, hardware transactions cannot commit as long as there is a concurrent fallback. This improves over HLE, but unfortunately has no safety guarantees. As was shown in the work by Dice et al. [12],

lock-removal may result in reading inconsistent memory states, executing illegal instructions, corrupting memory and more. However, we still provide results for lock-removal, as a reference that shows the potential of making this scheme work by providing the new hardware extensions proposed by Dice et al. [12].

4. *ALE:* Our new ALE scheme as described in Section 2 implemented into GCC.

3.1 Micro-benchmarks

We executed a set of micro-benchmarks on a red-black tree, hash-table and linked-list. The red-black tree is derived from the *java.util.TreeMap* implementation found in the Java 6.0 JDK. That implementation was written by Doug Lea and Josh Bloch. In turn, parts of the Java TreeMap were derived from Cormen et al. [9]. We implemented a standard linked-list, and use this list to implement a hash table, that is simply an array of lists. In addition, we introduce node padding to avoid false-sharing. We measured various padding lengths for small and large data-structure sizes, and found out that the overall best padding size is 16 longs (each long is 64bit).

All data-structures expose a key-value pair interface of *put*, *delete*, and *get* operations. If the key is not present in the data structure, put will put a new element describing the key-value pair. If the key is already present in the data structure, put will simply insert the value associated with the existing key. The get operation queries the value for a given key, returning an indication if the key was present in the data structure. Finally, delete removes a key from the data structure, returning an indication if the key was found to be present in the data structure.

Our benchmark first populates each data-structure to a predefined *initial-size*, and then executes put() and delete() with equal probability. For example, a *mutation ratio* of 10%, means that there is 5% put() and 5% delete(). We choose a random key for each operation from a *key-range* that is twice the size of the initial-size, so that mutations will actually mutate the data-structure. We report the average *throughput* of 3 runs, where each run executes for 10 seconds.

In top row of Figure 3, we present throughput results for a red-black tree with 1,000,000 nodes, and a hash-table with 262,144 nodes equally distributed over 8,192 buckets (approximately 32 nodes per bucket). We use a 20% mutation ratio for both data-structures. In the next two rows, we present an execution analysis, that reveals HTM (1) conflict, (2) capacity and (3) explicit abort ratios that occur in the fast-path, and the (4) fallback ratio, the relative amount of operations that completed execution in the fallback-path. The next three rows use the same format to present results for a linked-list with 100 nodes and 2% mutation ratio, and Kyoto CacheDB (details in Section 3.3). In these benchmarks, HLE and HLE-SCM exhibit similar performance so we plot only HLE.

As can be seen in Figure 3, ALE matches and significantly outperforms HLE in micro-benchmarks: for 16 threads, ALE is approximately 5.5-7 times faster than HLE for the red-black tree, hash-table and the linked-list. Notice that Unsafe-LR also provides significant improvements over HLE. The result of Unsafe-LR is interesting, and in some sense unexpected, since the concurrency that Unsafe-LR provides is limited (hardware cannot commit when there is a fallback). This shows that hardware exten-

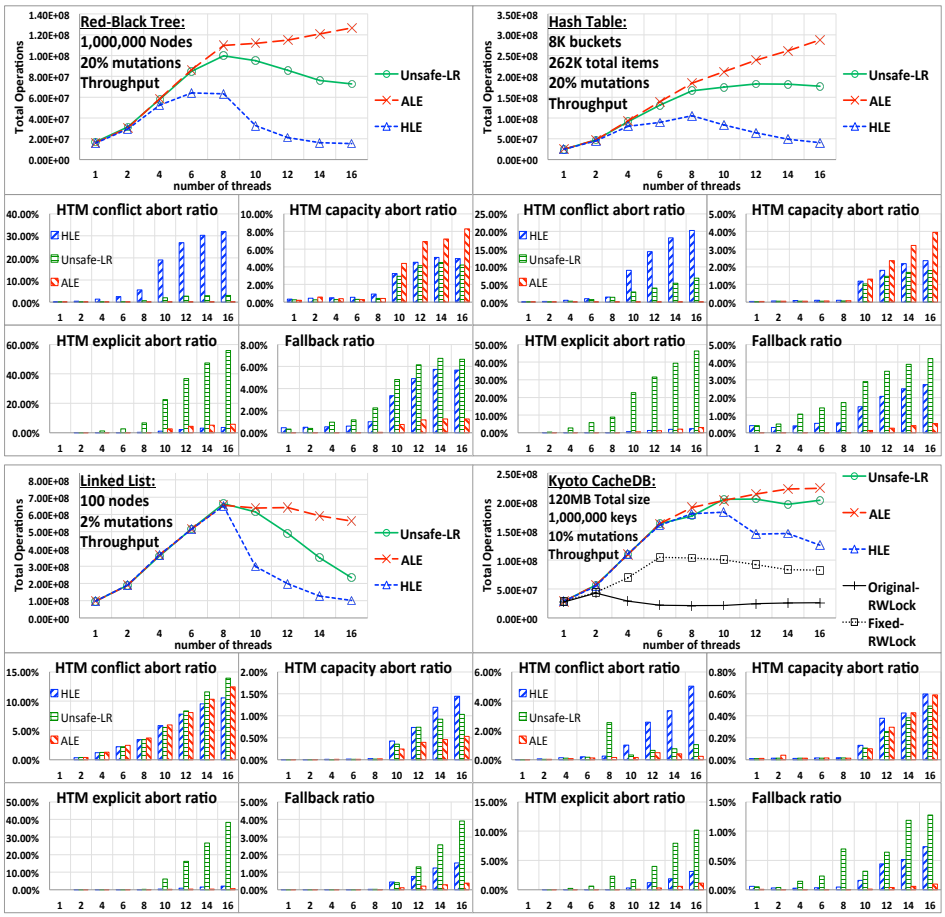

Fig. 3. Results for micro-benchmarks and Kyoto CacheDB

sions proposed by Dice et al. [12] (not present in current processors) to make Unsafe-LR fully safe will be beneficial in practice.

The main reason for the improvements of ALE is the full concurrency that it provides between fast-path hardware transactions and the fallback-path. In HLE, the fallback-path aborts all fast-path hardware transactions when it starts, which is why HLE exhibits a large amount of HTM conflict aborts (as can be seen in the HTM conflict abort ratio graphs). This is not the case in Unsafe-LR, that allows the fallback-path to proceed concurrently with fast-path hardware transactions. However, this concurrency is limited, because at commit-time fast-path hardware transactions must explicitly abort if there is a concurrent fallback. This generates a large amount of HTM explicit aborts (as can be seen in HTM explicit abort ratio graphs). In contrast, in ALE both HTM conflict and explicit abort ratios are very low (except for the linked list where the HTM conflicts

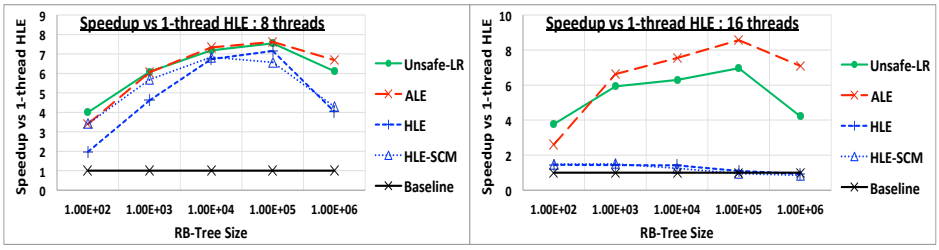

Fig. 4. Speedup results for 8 and 16 threads for various red-black tree sizes.

are a result of true contention). A side-effect of this is the reduced fallback-path ratios in ALE, which allow ALE to provide better results.

We also measured the success ratios of short hardware transactions that ALE uses in the fallback-path for segmentation and writeback. Our results show that the write-back succeeds to commit as a short hardware transaction 90-95% of the time. Also, the segmentation process works best for segment limits in the range of 20-40 shared reads/writes (per segment), so it reduces the lock barrier overheads by at least an order of magnitude.

3.2 Various Red-Black Tree Sizes

Figure 4 shows speedups of HLE, HLE-SCM, Unsafe-LR and ALE for 8 and 16 threads (from left to right) for various sizes of red-black tree. The baseline in these executions is 1-thread HLE.

We can see that ALE and Unsafe-LR are significantly faster than HLE and HLE-SCM for 16 threads. However, for 8 threads, these differences become smaller since less operations revert to execute in the fallback-path. This means that in order to see the advantages of ALE over HLE there should be a sufficient amount of fallbacks.

Notice that HLE-SCM is only beneficial over the standard HLE for small 100 nodes red-black tree on 8 threads. HLE-SCM reduces conflicts by serializing conflicting operations via an auxiliary lock, and since this case (of a small tree) is highly contended, it helps to reduce unnecessary conflicts and improve the overall performance.

3.3 KyotoCabinet

KyotoCabinet [18] is a suite of DBM data stores written in C++. In this benchmark, we focused on the in-memory component of the suite, called Kyoto CacheDB, that implements a bounded in-memory cache of key-value pairs where both the keys and values are opaque byte arrays. Internally, Kyoto CacheDB splits the database into slots, where each slot is a hash table of binary search trees. As a result, for each key, it first hashes the key into a slot, and then hashes again into a hash table of the slot. Next, it traverses the binary tree of the slot. The database ensures that the tree is bounded in size, by evicting entries that are least recently used (LRU policy). For synchronization, Kyoto

CacheDB uses a single coarse-grained read-write lock to start a database operation, and a per slot mutex lock to access a specific slot.

We replaced all locks of Kyoto CacheDB with ALE. Since our implementation of ALE uses GCC TM, the transformation is automated by the GCC that generates the necessary code paths and instrumentations. Our first comparisons of ALE to the original Kyoto CacheDB showed that the original read-write lock is a performance bottleneck, which concurs with the results of [13]. We also found that Kyoto performs an excessive amount of explicit thread context switches due to the specific implementation of reader-writer spin locks in the Linux pthreads library. Therefore, we replaced the original read-write lock of Kyoto CacheDB with an ingress-egress reader-writer lock implementation [14] that has no explicit context-switches. To the best of our knowledge, the ingress-egress reader-writer locks perform the best on Intel machines (ingress/enter counter and egress/exit counter for read-lock/read-unlock) [4]. We note that one could use hierarchical cohort-based reader-writer locks [15] in our benchmark to reduce the inter-thread cache traffic in Kyoto. However, this would not have a significant effect since the performance analysis reveals that the cache miss ratio is already low (4%-5%).

The benchmark for Kyoto CacheDB works in a similar way to our micro-benchmarks: it fills the database to a fixed initial size, and then executes gets/puts/deletes with random keys. Results are shown in Figure 3. We can see that ALE is twice faster than HLE. Notice, that this is not the same improvement like in the micro-benchmarks, where ALE was 5.5-7 times faster than HLE. The reason for this difference can be seen in the analysis: the HTM abort ratios are much lower for Kyoto CacheDB compared to the micro-benchmarks, which also results in low fallback ratios (1-2%). As a result, the reduction in HTM aborts that ALE provides is less dominant than in the micro-benchmarks, however, the ALE is still twice faster than HLE, and we believe that with increased concurrency it will become even more faster. Notice that Unsafe-LR is similar to ALE also due to low HTM aborts. However, in Unsafe-LR there is no safety guarantees and the program may crash, while ALE provides full safety.

4 Conclusion

We proposed *amalgamated lock-elision* (ALE), a new lock-elision scheme that provides concurrency between fast-path hardware transactions and the fallback-path, while preserving full safety guarantees. The key idea is to split the fallback-path into dynamic sections that fuse hardware and software with fine-grained locks in a way that provides efficiency. Our empirical results show that ALE is significantly faster than hardware lock elision (HLE) on both micro-benchmarks and a real use-case application, the Kyoto CacheDB. We believe that our results are encouraging, and show that hardware and software may be mixed in new and unexpected ways that were not originally intended by hardware and software designers.

5 Acknowledgments

We thank anonymous DISC referees for helpful and practical suggestions. This helped us to improve the paper and speed-up the algorithm. This work was supported by Israel Science Foundation under grant number 1386/11, National Science Foundation under grants CCF-1217921, CCF-1301926, and IIS-1447786, the Department of Energy under grant ER26116/DE-SC0008923, and the Intel Science and Technology Center in Big Data, Oracle and Intel corporations.

References

1. D. Dice A. Matveev and N. Shavit. Implicit privatization using private transactions. In *Transact 2010*, Paris, France, 2010.
2. T. Harris A. Roy, S. Hand. A runtime system for software lock elision. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 261–274, New York, NY, USA, 2009. ACM.
3. Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, and Justin Gottschlich. Draft specification of transactional language constructs for c++, 2012.
4. Yehuda Afek, Alexander Matveev, and Nir Shavit. Pessimistic software lock-elision. In Marcos K. Aguilera, editor, *DISC*, volume 7611 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2012.
5. Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 25:1–25:14, New York, NY, USA, 2014. ACM.
6. H. Attiya and E. Hillel. The cost of privatization. In *DISC*, pages 35–49, 2010.
7. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
8. Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. *SIGARCH Comput. Archit. News*, 41(3):225–236, June 2013.
9. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition edition, 2001.
10. M. Desnoyers, A. Stern P. McKenney, and J. Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 2009.
11. 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.
12. Dave Dice, Timothy L. Harris, Alex Kogan, Yossi Lev, and Mark Moir. Hardware extensions to make lazy subscription safe. *CoRR*, abs/1407.6968, 2014.
13. Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 188–197, New York, NY, USA, 2014. ACM.
14. Dave Dice and Nir Shavit. Tlrw: Return of the read-write lock. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 284–293, New York, NY, USA, 2010. ACM.
15. David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. *ACM Trans. Parallel Comput.*, 1(2):13:1–13:42, February 2015.

16. Nuno Diegues and Paolo Romano. Self-tuning intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 209–219, Philadelphia, PA, June 2014. USENIX Association.
17. Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 3–14, New York, NY, USA, 2014. ACM.
18. FAL Labs. Kyoto cabinet: A straightforward implementation of dbm, 2011.
19. Google. https://sites.google.com/site/tmforcplusplus, 2014.
20. T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.
21. Tim Harris and Keir Fraser. Concurrent programming without locks.
22. Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007.
23. M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.
24. Calciu Irina, Shpeisman Tatiana, Pokam Gilles, and Herlihy Maurice. Improved single global lock fallback for best-effort hardware transactional memory. In *Transact 2014 Workshop*, 2014.
25. V. Marathe, M. Spear, and M. Scott. Scalable techniques for transparent privatization in software transactional memory. *Parallel Processing, International Conference on*, 0:67–74, 2008.
26. A. Matveev and N.Shavit. Reduced hardware transactions: a new approach to hybrid transactional memory. In *SPAA*, pages 11–22, 2013.
27. Alexander Matveev and Nir Shavit. Reduced hardware norec: A safe and scalable hybrid transactional memory. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2015, Istanbul, Turkey, 2015. ACM.
28. Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic stm. In *Transact 2008 Workshop*, 2008.
29. C. Fetzer P. Felber and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, New York, NY, USA, 2008. ACM.
30. R. Rajwar and J. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO*, pages 294–305. ACM/IEEE, 2001.
31. Web. Intel tsx http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell, 2012.
32. Afek Yehuda, Levy Amir, and Morrison Adam. Software-improved hardware lock elision. In *PODC 2014*, Paris, France, 2014. ACM Press.