acmqueue

# Scaling Existing Lock-based Applications with Lock Elision

**Lock elision enables existing lock-based programs to achieve the performance benefits of nonblocking synchronization and fine-grain locking with minor software engineering effort.**

Andi Kleen, Intel Corporation

Multithreaded applications take advantage of increasing core counts to achieve high performance. Such programs, however, typically require programmers to reason about data shared among multiple threads. Programmers use synchronization mechanisms such as mutual-exclusion locks to ensure correct updates to shared data in the presence of accesses from multiple threads. Unfortunately, these mechanisms serialize thread accesses to the data and limit scalability.

Often, lock-based programs do not scale because of the long block times caused by serialization, as well as the excessive communication overhead to coordinate synchronization. To reduce the impact on scalability, programmers use fine-grained locking, where instead of using a few locks to protect all shared data (coarse granularity), they use many locks to protect data at a finer granularity. This is a complex and error-prone process[10,11] that also often impacts single-thread performance.

Extensive work exists on ways to improve synchronization performance. Lock-free data structures support concurrent operations without mutually exclusive locks.[12] Such algorithms are often quite complex.

A rich variety of locking algorithms of varying complexity exist.[10] Other approaches such as optimistic locking avoid synchronization overhead—for example, by using sequence numbers to protect reading shared data and retrying the accesses if necessary. While effective under certain conditions, extending these approaches to be generally usable is quite difficult.

Transactional memory[5] proposed hardware mechanisms to simplify the development of lock-free data structures. They rely on mechanisms other than locks for forward progress and exploit the underlying cache-coherence mechanisms to detect conflict among threads.

Lock elision[14] was another proposal to expose concurrency in lock-based programs by executing them in a lockless fast path. It uses the hardware capability of modern processors and the underlying cache-coherence protocol to execute critical sections optimistically, without acquiring a lock. The lock is acquired only when actually required to resolve a data conflict.

In spite of the numerous proposals, high-performance synchronization remains difficult: programmers must use information known in advance to determine when to serialize, and scalable locking is complex, leading to conservative lock placement.

Recently, commercial processors from Intel Corporation and IBM have introduced hardware support to improve synchronization.[6,7,8,16] This presents a unique opportunity for software developers to improve the scalability of their software.

This article focuses on improving the existing lock-based programming model, and thus, looks at lock elision as the primary usage model.

## HARDWARE SUPPORT FOR LOCK ELISION

Programmers must decide at development time how to control access to shared data—whether

by using coarse-grained locks, where a few locks protect all data, or by using fine-grained locks to protect different data. The dynamic behavior eventually determines sharing patterns. Finding errors with incorrectly used synchronization is quite difficult.

What is needed is a mechanism that has the usability of coarse-grained locks with the scalability of fine-grained locks. This is exactly what lock elision provides. The programmer must still use locks to protect shared data but can adopt a more coarse-grained approach. The hardware determines dynamically whether threads need to serialize in a critical section and executes the lock-based programs in a lock-free manner, where possible.

The processor executes lock-protected critical sections (called transactional regions) transactionally. Such an execution only reads the lock; it does not acquire or write to it, thus exposing concurrency. Because the lock is elided and the execution optimistic, the hardware buffers any updates and checks for conflicts with other threads. During the execution, the hardware does not make any updates visible to other threads.

On a successful transactional execution, the hardware atomically commits the updates to ensure all memory operations appear to occur instantaneously when viewed from other processors. This atomic commit allows the execution to occur optimistically without acquiring the lock; the execution, instead of relying on the lock, now relies on hardware to ensure correct updates. If synchronization is unnecessary, the execution can commit without any cross-threaded serialization.

A transactional execution may be unsuccessful because of abort conditions such as data conflicts with other threads. When this happens, the processor will do a transactional abort. This means the processor discards all updates performed in the region, restores the architectural state to appear as if the optimistic execution never occurred, and resumes execution nontransactionally. Depending on the policy in place, the execution may retry lock elision or skip it and acquire the lock.

To ensure an atomic commit, the hardware must be able to detect violations of atomicity. It does this by maintaining a read and a write set for the transactional region. The read set consists of addresses read from within the transactional region, and the write set consists of addresses written to, also from within the transactional region.

A conflicting data access occurs if another logical processor reads a location that is part of the transactional region's write set or writes a location that is a part of the read or write set of the transactional region. This is referred to as a *data conflict*.

Transactional aborts may also occur as a result of limited transactional resources. For example, the amount of data accessed in the region may exceed the buffering capacity. Some operations that cannot be executed transactionally, such as I/O, always cause aborts.

## INTEL TSX

Intel TSX (Transactional Synchronization Extensions) provides two instruction-set interfaces to define transaction regions. The first is HLE (Hardware Lock Elision), which uses legacy-compatible instruction prefixes to enable lock elision for existing atomic-lock instructions. The other is RTM (Restricted Transactional Memory), which provides new XBEGIN and XEND instructions to control transactional execution and supports an explicit fallback handler.

Programmers who want to run Intel TSX-enabled software on legacy hardware could use the HLE interface to implement lock elision. On the other hand, with more complex locking primitives or when more flexibility is needed, the RTM interface can be used to implement lock elision.

This article focuses on the RTM interface in TSX. IBM systems provide instructions roughly similar to RTM.[6,8,16]

### FALLBACK HANDLERS

To use the RTM interface, programmers add a lock-elision wrapper to the synchronization routines. Instead of acquiring the lock, the wrapper uses the XBEGIN instruction and falls back to the lock if the transaction aborts and retries don't succeed.

The Intel TSX architecture (and others, except the IBM zSeries, which has limited support for guaranteed small transactions[8]) does not guarantee that a transactional execution will ever succeed. Software must have a fallback nontransactional path to execute on a transactional abort. Transactional execution does not directly enable new algorithms but improves the performance of existing ones. The fallback code must have the capability of ensuring eventual forward progress; it must not simply keep retrying the transactional execution forever.

To implement lock elision and improve existing lock-based programming models, the programmer would test the lock within the transactional region and acquire the lock in the nontransactional fallback path, and then reexecute the critical region. This enables a classic lock-based programming model.

FIGURE 1

**Lock Elision**

```
Listing:
/* Start transactional region. On abort we come back here. */
if (_xbegin() == _XBEGIN_STARTED) {
       /* Put lock into read-set and abort if lock is busy */
       if (lock variable is not free)
              _xabort(_XABORT_LOCK_BUSY);
} else {
       /* Fallback path */
       /* Come here when abort or lock not free */
       lock lock;
}
/* Execute critical region either transaction or with lock */

Elided lock

Listing:
/* Critical region ends */
/* Was this lock elided? */
if (lock is free)
       _xend();
else
       unlock lock
Elided unlock
```

Alternatively, the programmer may use the transactional support to improve the performance and implementation of existing nonblocking heavyweight algorithms, using a fast and simple path to execute transactionally, but a slower and more complex nonblocking path if the transactional execution aborts. The programmer must ensure proper interaction between the two paths.

The programmer may also use the transactional support for implementing new programming models such as transactional memory. One approach uses the transactional support in hardware for a fast path, with an STM (software transactional memory) implementation for the fallback path[2] if the large overhead of STM on fallback is acceptable.[1]

### IMPLEMENTING LOCK ELISION

Lock elision uses a simple transactional wrapper around existing lock code, as shown in figure 1.

`_xbegin()` tries to execute the critical section transactionally. If the execution does not commit, then `_xbegin` returns a status word different from `_XBEGIN_STARTED`, indicating the abort cause. After doing some retries, the program can then acquire the lock.

On unlock, the code assumes that if the lock is free, then the critical region was elided, and it commits with `_xend()`. (This assumes that the program does not unlock free locks.) Otherwise, the lock is unlocked normally. This is a simplified (but functional) example. A practical implementation would likely implement some more optimizations to improve transaction success. GCC (GNU Compiler Collection) provides detailed documentation for the intrinsics used here.[4] GitHub has a reference implementation of the TSX intrinsics.[9]

A thread that keeps aborting and goes to the fallback handler eventually acquires the lock. All threads speculating on the same lock have the lock in their read sets and abort their transactional execution when they detect a write conflict on the lock variable. If that happens, then eventually they will all take the fallback path. This synchronization mechanism between fallback path and speculative execution is important to avoid races and preserve the locking semantics.

This also requires that the lock variable is visible to the elision wrapper, as shown in figure 2.

FIGURE 2

**Example Synchronization of Transactions through the Lock Variable**

```
Thread 1                    Thread 2
Transaction A
Read Lock X
                            Transaction B
                            Read Lock X
                            … abort for some reason …
                            Fallback handler
                            Writes to lock X to acquire
                    * DATA CONFLICT ON LOCK *
Abort
Waits on lock to become free
```

#### ENABLING LOCK ELISION IN EXISTING PROGRAMS

Most existing applications use synchronization libraries to implement locks. Adding elision support to only these libraries is often sufficient to enable lock elision for these applications. This may be as simple as adding an elision wrapper in the existing library code. The application doesn't need changes for this step, as lock elision maintains the lock-based programming model.

We implemented lock elision for the POSIX standard pthread mutexes using Intel TSX and integrated it into Linux glibc 2.18. An implementation to elide pthread read/write locks has not yet been integrated into glibc.

The glibc mutex interface is binary compatible. This allows existing programs, using pthread mutexes for locking, to benefit from elision. Similarly, other lock libraries can be enabled for elision.

The next step is to evaluate performance and analyze transactional execution. Such an analysis may suggest changes to applications to make them more elision-friendly. Such changes also typically improve performance without elision, by avoiding unnecessary communication among cores.

#### ANALYZING TRANSACTIONAL EXECUTION

The behavior of explicit speculation is different from existing programming models. An effective way of analyzing transactional execution is with a transaction-aware profiler using hardware performance-monitoring counters.

Intel TSX provides extensive support to monitor performance, including sampling, abort rates, abort-cause profiling, and cycle-level analysis for successful and unsuccessful transactional executions. Often, the abort rates can be significantly reduced through minor changes to application data structures or code (for example, by avoiding false sharing).

The performance-monitoring infrastructure provides information about hardware behavior that is not directly visible to programmers. This is often critical for performance tuning. Programmers can also instrument transactional regions to understand their behavior. This provides only a limited picture, however, because transactional aborts discard updates, and the instrumentation itself may contribute to aborts.

### ELISION RESULTS

Figure 3 compares the average number of operations per second when reading a shared hash table protected with a global lock versus an elided global lock. The tests were run on a four-core Intel Core (Haswell) system without SMT. The elided lock provides near-perfect scaling from one to four cores, while the global lock degrades quickly. A recent paper[17] analyzes lock elision using Intel TSX for larger applications and reports compelling benefits for a wide range of critical section types.

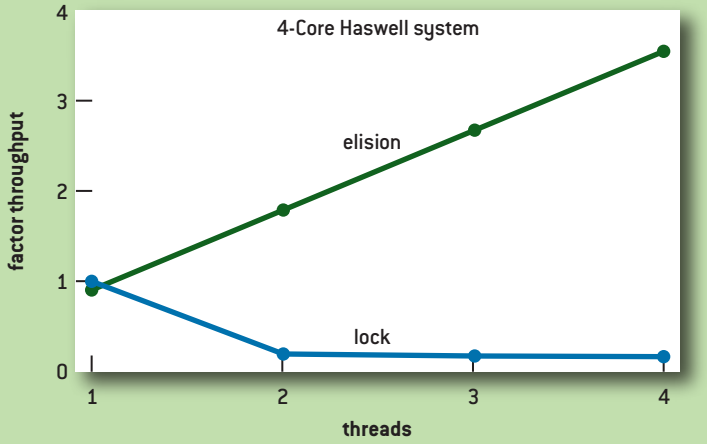
#### WHEN DOES ELISION HELP?

Data structures that do not encounter significant data conflicts are excellent candidates for elision. An elided lock can allow multiple readers and nonconflicting writers to execute concurrently. This is because lock elision does not result in any write operations on the lock, thus behaving as a reader-writer lock. It also eliminates any cache-line communication of the lock variable. As a result, programs with fine-grained locking may also see gains.

Not all programs benefit from lock improvements. These programs are either single-threaded, do not use contended locks, or use some other algorithm for synchronization. Data structures that repeatedly conflict do not see a concurrency benefit.

FIGURE 3

**Hash Table Lookup with Elision**




If the program uses a complex nonblocking or fine-grained algorithm, then a simpler transactional fast path can speed it up. Further, some programs using system calls or similar unfriendly operations inside their critical sections will see repeated aborts.

### THE COST OF ABORTS

Not all transactional aborts make the program slower. The abort may occur where otherwise the thread would simply have been waiting for a lock. Such transactional regions that abort may also serve to prefetch data. Frequent, persistent aborts hurt performance, however, and developers must analyze aborts to reduce their probability.

### ADAPTIVE ELISION

Synchronization libraries can adapt their elision policies according to transactional behavior. For example, the glibc implementation uses a simple adaptive algorithm to skip elision as needed. The synchronization library wrapper detects transactional aborts and disables lock elision on unsuccessful transactional execution. The library reenables elision for the lock after a period of time, in case the situation has changed. Developing innovative adaptive heuristics is an important area of future work.[13,15]

Adaptive elision combined with elision wrappers added to existing synchronization libraries can effectively enable lock elision seamlessly for all locks in existing programs. Developers should still use profiling to identify causes of transactional aborts, however, and address the expensive ones. That remains the most effective way to improve performance.

As an alternative to building adaptivity into the elision library, programmers may selectively identify which locks to apply elision to. This approach may work for some applications but may require developers to understand the dynamic behavior of all the locks in the program, and it may result in missed opportunities. Both approaches can be combined.

### THE LEMMING EFFECT

When a transactional abort occurs, the synchronization library may either retry elision or explicitly skip it and acquire the lock. How the library retries elision is important. For example, when a thread falls back to explicitly acquiring a lock, this results in aborting other threads that elide the same lock. A naïve implementation on the other threads would immediately retry elision. These threads would find the lock held, abort, and retry elision, thus quickly reaching their retry threshold without any opportunity to have found the lock free. As a result, these threads quickly transition to an execution where they skip elision. It is easy to see how all threads end up in a situation where no thread reattempts lock elision for longer time periods. This phenomenon is the lemming effect[3] where threads enter extended periods of nonelided execution. This prevents software from taking advantage of the underlying elision hardware.

A simple and effective fix in the synchronization library is to retry elision only if the lock is free and spin/wait nontransactionally—and limit retry counts. Some lock algorithms implement a queue to enforce an order for threads if they find the lock unavailable. Queues are incompatible with parallel speculation. For such code, programmers must use the elision wrapper, including retry support, prior to the actual queuing code. Other mitigation strategies are possible.

Once programmers understand the underlying behavior, simple fixes can go a long way in improving unexpected performance behaviors.

Lock elision is a new technique in the scaling toolbox that is available on modern systems. It enables existing lock-based programs to achieve the performance benefits of nonblocking synchronization and fine-grained locking with minor software engineering effort.

### ACKNOWLEDGMENTS

Thanks to Noel Arnold, Jim Cownie, Roman Dementiev, Ravi Rajwar, Arch Robison, Joanne Strickland, and Konrad Lai for their feedback and improvements to the article.

### FURTHER READING

Intel. *IA Optimization Manual*, Chapter 12, TSX Recommendations, describes various tuning techniques to improve lock elision; http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.
Web Resources: http://www.intel.com/software/tsx.

### REFERENCES

1. Cascaval, C., Blundell, C., Michael, M., Cain, H. W., Wu, P., Chiras, S., Chatterjee, S. 2008. Software transactional memory: why is it only a research toy? *Communications of the ACM* 51(11): 40-46.
2. Dalessandro, L., Carouge, F., White, S., Lev, Y., Moir, M., Scott, M. L., Spear, M. F. 2011. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In the *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*: 39-52.
3. Dice, D., Herlihy, M., Lea, D., Lev, Y., Luchangco, V., Mesard, W., Moir, M., Moore, K., Nussbaum, D. 2008. Applications of the adaptive transactional memory test platform. The 3rd Annual ACM SIGPLAN Workshop on Transactional Computing.

4. GCC. 2013. X86 transaction memory intrinsics; http://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/X86-transactional-memory-intrinsics.html#X86-transactional-memory-intrinsics.
5. Herlihy, M., Moss, J. E. B. 1993. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA):* 289-300.
6. IBM. 2013. Power ISA; http://www.power.org/documentation/power-isa-version-2-07/.
7. Intel. 2012. *Intel 64 and IA-32 Architectures Software Developer Manuals*, Volume 1, Chapter 14; http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.
8. Jacobi, C., Slegel, T., Greiner, D. 2012. Transactional memory architecture and implementation for IBM System z. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture:* 25-36.
9. Kleen, A. 2013. tsx-tools; http://github.com/andikleen/tsx-tools.
10. McKenney, P. E. 2013. Is parallel programming hard, and, if so, what can you do about it?; https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html
11. McVoy, L. 1999. SMP scaling considered harmful; http://www.bitmover.com/llnl/smp.pdf.
12. Michael, M. M. 2013. The balancing act of choosing nonblocking features. *Communications of the ACM* 56(9): 46-53.
13. Pohlack, M., Diestelhorst, S. 2011. From lightweight hardware transactional memory to lightweight lock elision. In the Sixth Annual ACM SIGPLAN Workshop on Transactional Computing.
14. Rajwar, R., Goodman, J. R. 2001. Speculative lock elision. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture:* 294–305.
15. Usui, T., Behrends, R., Evans, J., Smaragdakis, Y. 2009. Adaptive locks: combining transactions and locks for efficient concurrency. In the 4th ACM SIGPLAN Workshop on Transactional Computing.
16. Wang, A., Gaudet, M., Wu, P., Amaral, J. N., Ohmacht, M., Barton, C., Silvera, R., Michael, M. 2012. Evaluation of Blue Gene/Q hardware support for transactional memory. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques:* 127-136.
17. Yoo, R. M., Hughes, C. J., Rajwar, R., Lai, K. 2013. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*: article no. 19.

**LOVE IT, HATE IT? LET US KNOW**

feedback@queue.acm.org

**ANDI KLEEN** is a software engineer at Intel's open source technology center. He worked on the x86-64 port of the Linux kernel and other kernel areas, including networking, performance, NUMA (nonuniform memory access), and error recovery. He currently focuses on performance tuning, scalability to many cores, and performance analysis. He started his career doing support for media artists and then spent more than nine years in SUSE Labs.