From Lightweight Hardware Transactional Memory to Lightweight Lock Elision

Martin Pohlack  Stephan Diestelhorst
Advanced Micro Devices, Inc.
ASF_Feedback@amd.com

Abstract
AMD’s Advanced Synchronization Facility (ASF) has been evaluated in earlier work in the context of hardware and hybrid transactional memory, software transactional memory, and lock-free programming. In this work, we describe an extension to ASF for applying it in the area of lock elision (LE), which is now a well established concept in academia, but has not found its way into mainstream micro-processors.

We extended ASF to allow transactional execution of unmodified binary code, minimizing toolchain requirements and employing this extension to run existing lock-base multithreaded programs using a combined software-hardware approach. Software is responsible for demarcating transaction boundaries, for register snapshotting, for providing an elision policy, and a software backup path. Hardware in the form of an extended ASF is used for data conflict detection at runtime and rolling back modified memory in an abort case.

Early measurement results for a memcached-based setup show great potential for concurrent execution.

1. Introduction
The ubiquity of commodity multi-core processors and diminishing performance gains for single-core processors have spurred interest in making parallel programming more applicable to a wider variety of applications and tangible for a larger number of programmers. Transactional memory [19] proposes simple semantics of executing blocks of code atomically, but with enabled fine-grained parallelism. As such, transactional memory requires changes to the source code of applications and elaborate adaptations to the compiler and hardware. Although the original goal of transactional memory has been simple semantics, it turns out that software transactional memory (STM) implementations provide a wealth of weakened atomicity semantics [32] and pay a performance premium to support stronger semantics. The weakened semantics in turn compromise the simplicity goal by requiring knowledge of the underlying STM algorithm.

Although hardware solutions can provide stronger semantics, they suffer from limitations induced by the underlying microarchitecture. Existing attempts to lift these limitations complicate the microarchitecture [5, 7, 9, 22, 29], and consequently have not been adopted by any of the industry-proposed transactional memory proposals.

Speculative lock elision (SLE) [27] re-uses the existing locking infrastructure and critical-section annotations and executes non-conflicting critical sections in parallel. To be transparent on the instruction set architecture (ISA) level, SLE requires prediction and tracking logic in addition to the HTM-like speculation logic. In the AMD64 ISA, the efforts of the prediction and detection logic are complicated by the many different ways to write locks and also by several idioms using the same instructions but not demarcating critical sections (statistics updates, lock-free data structures). SLE misprediction may cause significant numbers of wasted cycles due to aborts and subsequent re-tries of code not belonging to critical sections.

In this paper, we propose to lift the simplicity requirement of transactional memory, and instead offer incremental performance improvements for incremental programmer effort, akin to earlier work on transactional lock elision (TLE) [13]. We also propose to reduce the burden on the hardware prediction and tracking logic and reduce erroneous speculation attempts by keeping entry into and exit from speculation as special instructions. For this we extend AMD’s Advanced Synchronization Facility (ASF, [4]) with a speculation-by-default mode, allowing execution of unmodified code inside the critical sections. We effectively side-step the need for re-compilation by wrapping the pthread-library mutex functionality with support for lock elision using our new instructions. This wrapper library can be linked dynamically (at load time) to unmodified, binary applications and makes the benefits of LE available to them. Using this infrastructure, we will report on early results with a memcached-based setup in this paper.

2. AMD’s Advanced Synchronization Facility
AMD’s ASF [4] is a flexible, multi-word atomic primitive, much like transactional memory. To keep micro-
architectural complexity small, we have opted for an almost entirely best-effort design: The processor is free to abort any ongoing transaction at any time. In particular, data conflicts, capacity overflows, exceptions, interrupts, and specific unsupported instructions can lead to an abort. However, in the absence of conflicts and when keeping the number of accessed locations within the worst-case capacity bound, \( \text{ASF} \) will not indefinitely abort transactions; the intention is to have transactions succeed on the first try most of the time.

\( \text{ASF} \) tracks data at cache-line granularity (naturally aligned blocks of 64 bytes) and aborts in-flight transactions with conflicts in the tracked working sets, employing a simple requester-wins abort policy. Conflicts are also detected between transactions and normal code, making \( \text{ASF strongly isolating} \).

\( \text{ASF} \) extends the AMD64 ISA with seven new instructions: \text{SPECULATE} and \text{COMMIT} begin and end transactions; \text{LOCK MOV} (both loads and stores) is used for speculative data access and conflict detection, while \text{WATCHR} and \text{WATCHW} arm the conflict-detection mechanism without loading the data. The \text{RELEASE} instruction will remove an unmodified cache line from the set of conflict-checked locations. Finally, \text{ABORT} allows code within a transaction to voluntarily abort it.

An abort in \( \text{ASF} \) is performed by rolling back modifications to cache lines accessed with \text{LOCK MOV} or protected with \text{LOCK WATCHR/W}, reverting the stack pointer to the value it had when passing \text{SPECULATE}, returning an abort status code in the rAX and rFLAGS registers, and finally resetting the instruction pointer to the instruction following \text{SPECULATE}. Note that \( \text{ASF} \) does not keep a full register snapshot, nor does it explicitly have the notion of an abort handler. The former can easily be achieved by register clobbering code from the compiler, and the latter can be emulated by checking the abort code after \text{SPECULATE} (which is zero on success and non-zero on an abort) and branching to an appropriate handler.

Aborts caused by transient conditions, such as interrupts or page faults due to lazy paging, and conflicts, usually warrant a re-try of the transaction, assuming that the transient condition has vanished in the meantime. To this end, \( \text{ASF} \) conveys status information in the abort code, allowing streamlined re-try logic in the application. In addition, \( \text{ASF} \) makes exceptions inside transactions visible after the abort so that page faults can be handled by the operating system, transparently to user code.

The \( \text{ASF} \) specification does not mandate a particular implementation; we have experimented with a number of them in earlier publications \([6, 8, 10, 14, 16]\).

**Selective annotation** Transactions in \( \text{ASF} \) do not mandate each memory access to be annotated with the \text{LOCK} prefix, but instead also allows normal \text{MOV} and other memory accessing instructions. These instructions will perform standard, non-speculative memory operations, but will not add the accessed memory location (the accessed cache line(s)) to the set of tracked locations causing abort on conflict. This **selective annotation** allows significant reductions in the footprint of applications \([10]\), for example, by removing thread-local or stack accesses from the limited working set. Selective annotation can be employed by compilers automatically, proving that specific locations are thread local, or by expert programmers tweaking performance through working-set size and conflict probability reductions \([12, 14, 30]\).

### 3. Speculation by Default

\( \text{ASF} \) in its existing shape is targeted to be used with strong toolchain support for making the most use of its limited hardware capacity. The selective annotation feature is a key component here and can be used to great benefit given a suitable toolchain \([6]\).

However, due to the need for affixing \text{LOCK} prefixes to \text{MOV} instructions for speculative memory accesses, it does not support execution of unmodified binary code within a transaction. Code executed that way would issue non-speculative memory instructions, which would not participate in conflict detection and would not be rolled back on abort.

To fulfill our transparency requirement for \( \text{LE} \), we decided to introduce a new \( \text{ASF} \) mode, *speculation by default*, which changes the "polarity" of the \text{LOCK} prefix annotation: accesses with the prefix become non-speculative, while all other accesses are treated as speculative within transactions. For starting speculation-by-default transactions, we provide a new instruction, \text{SPECULATE_INV}.

The high-level effect of this inversion is that high-level language code and third-party library code can be called directly inside transactions. No additional compilation or instrumentation step is required as *all standard memory accesses* become part of the enclosing transaction and are subject to conflict detection and rollback. Toolchain support required to work with this mode is minimal; usually, only some assembly-language bindings with register clobbering are required for forcing the compiler to do the snapshotting. Of course, some of \( \text{ASF} \)'s limitations are still active: hardware capacity is limited, certain instructions are still illegal inside transactions, and far calls (e.g., kernel entries due to interrupts or system calls) will still lead to aborts.

This new mode lends itself to a software-supported implementation of lock elision.

### 4. Application Example

For evaluating \( \text{ASF} \)-based lock elision, we looked at lock-based multi-threaded workloads that have potential for speculative execution of concurrent critical sections. In this section, we report on our experience with memcached, which...
4.1 memcached

Memcached is a distributed key-value in-memory database. In a large setup, several server instances run in a cluster-like environment serving from in-memory hash tables. Clients distribute load across the server instances by hashing the key part of requests [13]. Memcached is used, for example, by Facebook, Flickr, Twitter, and YouTube.

For our experiments we used the most recent public version from memcached.org (version 1.4.5). This version also supports running with multiple threads and supports the new, binary version of the memcached protocol [1].

The two most simple commands in the memcached protocol are GET and PUT for requesting and storing and string under a given key, respectively. GET-type requests are usually assumed to be the most common ones for memcached setups.

Memcached internally uses several locks for protecting critical data structures against races by concurrently running threads. We extended mutrace [25] to also measure blocking time per mutex to help identifying potentially contending locks. In our experiments, the central cache_lock was by far the most contended one, suggesting around 20% blocking time in the server for a high-load scenario. Other locks are used for protecting access to statistics and memory allocator data structures. All locks seen in memcached are standard pthread mutexes.

After considering the typical situation that GET requests usually dominate the workload, one is tempted to replace the standard mutexes with reader-writer locks. From looking into the code, it turns out that even typical read requests (like GET) do not follow pure read paths in the server. Occasionally, statistics have to be written, living timeouts for entries trigger, or the hashtable has to be resized. Typical GET paths are not read-only, but are read-mostly.

A second observation is that locks are usually only held for very brief moments. Typically, only some pointers are exchanged with brief meta-data updates. There is no memcpy() or similar code in the critical sections that would depend on the actual data targeted.

To summarize: (a) critical sections protected by the cache_lock are pessimistic in the sense that often only read-access is required and concurrent readers would typically be possible, and (b) those critical sections are usually very short, with few memory locations touched, lending themselves to a hardware-based solution with limited capacity.

4.2 Setup

Our complete measurement setup is located in one instance of the full-system simulator PTLsim, which we enhanced with ASF implementations [6] and the extension speculation by default. The implementation of the extension in PTLsim is relatively straightforward: The speculative bit for memory micro-ops is inverted in an early pipeline stage. The most tricky part here was not to invert several times upon potential replay of instructions. We configured PTLsim to simulate a machine with eight cores and adapted the memory latencies and the core model to be similar to those obtained on native AMD Opteron processors of families 0Fh (K8 core) and 10h (formerly code-named “Barcelona”).

We run both the client workload generator and the memcached server inside this single machine and use four threads for each. Although a multi-machine setup would be more realistic, this approach is the only one feasible when using the PTLsim full-system simulator. PTLsim can only simulate a single machine at once and does not have support for simulation of a full networking infrastructure. Also, coupling a simulated server with a non-simulated client machine via network is infeasible, due to the huge slow-down factor in the simulator that would lead to network timeouts. In a way, this setup behaves like a worst-case scenario for the memcached server, because network latency is very small and bandwidth is not limited by a physical network.

For generating workloads we use memslap, which is very suitable for finding sustained maximum throughput for a given setup. It contains a custom modern implementation of the memcached protocol and supports the modern binary version of the protocol and many concurrent requests. Memslap is part of the standard client library libmemcached. In its default configuration, memslap uses 90% GET requests and 10% PUT requests. The default key size is 64 bytes and the default value size is 1024 bytes. We use this default configuration unless otherwise noted.

We experimented with each memslap parameter in a native, gigabit-network setup to determine values that would maximize system throughput to put high contention on the mutexes inside memcached. As a result of these experiments, we use a window size of 10 000 for each concurrency, with 256 concurrent requests. Memslap uses 90% GET requests and 10% PUT requests. The default key size is 64 bytes and the default value size is 1024 bytes. We use this default configuration unless otherwise noted.

We configured PTLsim to simulate a server at our local test machine and use the binary protocol for communication. A typical command line in our experiments looks like this: memslap -w10k -c256 -T4 -t500000 -s localhost -B.

4.3 Manual instrumentation

As a first approach, we manually replaced the locking code in the very common GET path of memcached (item_get() in thread.c) with ASF-based lock elision code (cf. to Figure[1]). item_get() is a wrapper around do_item_get() that grabs the main cache_lock for the hash table. From a high-level point of view, pthread_mutex_lock is the point to start the ASF speculation and pthread_mutex_unlock is replaced with ASF’s commit instruction. Starting the speculation comprises enforcing a register snapshot by the compiler by clobbering all relevant registers inside an inline assem-
item *item_get(const char *key, const size_t nkey) {
    item *it;  int retries = 5;  ulong asf_fail;

    while (retries > 0) { // ASF lock-elision path
        asf_speculate_inv(asf_fail); // start speculation with inverted semantics
        if (unlikely (asf_fail)) { // rollback point
            if (asf_hard_error(asf_fail)) { // hard error, don't try again
                retries = 0;
            } else { // soft error, maybe try again
                retries--;
            }
            continue;
        }
        if (cache_lock.__data.__lock) { // pull lock into ASF's readset and check for race
            asf_abort(1); // lock was already held, bail out
        }
        it = do_item_get(key, nkey); // actual memcached GET path
        asf_commit_(); // commit ASF transaction
        return it;
    }

    pthread_mutex_lock(&cache_lock); // software fall-back path: really grab lock
    it = do_item_get(key, nkey); // ...
    pthread_mutex_unlock(&cache_lock); // ...
    return it; // ...
}

Figure 1. Simplified manual instrumentation in GET path.

Figure 2. Software predictor state machine with the hardware lock-elision level stored per mutex and thread. Low levels incur a high chance to try hardware elision; high-levels imply a very low probability. Failed hardware attempts increase the level; successful ones decrease the level.

ble snippet and adding the actual lock part of the pthread_mutex struct to the transaction’s read set by reading it. After the transaction is started, one has to look into the lock variable to verify that it was actually free at the time of starting the transaction. A side-effect of looking into the variable is its addition into the read set.

In case of contention or aborts, we currently employ a very simplistic approach of retrying a small number of times with ASF before falling back to the traditional approach of actually taking the lock without ASF. The write request to the actual lock done inside pthread_mutex_lock in the fall-back path is the point that aborts potentially running ASF transactions on other cores.

4.4 Dynamic instrumentation

Obviously, manual instrumentation has some disadvantages: access to source code is required and potentially a lot of locations have to be patched. To eliminate these problems, we use a technique of wrapping access to functions in shared libraries. We designed a library that wraps calls to pthread_mutex_lock and -unlock using Linux’s LD_PRELOAD mechanisms. The core approach to elide locks is very similar to the manual instrumentation approach discussed in Section 4.3, but we now generate statistics about each mutex that we use to better guide the decision whether lock elision is feasible for a given mutex at a given time.

We count the number of soft and hard aborts, the number of successful ASF transactions, and the number of normal lock operations required. Our software predictor state machine represents a notion of recent elision successes per mutex and thread and is comprised of a level and a chance for hardware re-try (cf. to Figure 2). The level is increased on ASF transaction failures, and decreased on successful ASF transactions. The current level for a given mutex determines the chance to actually try to elide the lock using ASF. For low levels, the chance is high and for high levels, the chance is very low (for level 0 mutexes, ASF will be always attempted; level 10 mutexes very rarely attempt ASF elisions but often directly fall back to normal mutex operations).
currently implement the chance with a variable, counting normal locking approaches. If this counter reaches a level-specific threshold, another ASF attempt is made.

That way, mutexes quickly adapt to the actual code protected by them at runtime and are still able to adjust to changes in workload.

In Section 4.5 we provide results for two slightly different strategies for reducing the level part of the statistics. The more aggressive strategy starts a hardware transaction directly after a level reduction, while the more conservative strategy first runs a level-specific amount of software locking rounds. The aggressive strategy is able to adapt more quickly to hardware-lock-elision-friendly changes but might overshoot doing so.

The dynamic approach that we describe here also incurs additional overhead of two forms: (a) call indirection and (b) statistics gathering. We user another level of indirection for acquiring locks. Where the manual instrumentation approach could directly inline the ASF code into the memcached code, the dynamic instrumentation puts this code into a function in a shared library, which itself does another indirect call to the actual pthread_mutex functions in the fallback case.

The dynamic part also gathers and uses per-mutex thread-local statistics, which are stored in a hash-table indexed by mutexes’ addresses. This information is updated after transactions and guides the lock elision process before transactions. For the manual instrumentation approach, we simply use one static policy for the single case that we support.

The results show that the additional effort done with the dynamic instrumentation pays off in the form of increased throughput.

4.5 Results

Although we would like to report on a huge number of experiments, running everything inside a full-system simulator severely limits our resources to do so. Simulations inside PTLsim typically run six to seven orders of magnitude slower than on bare metal. We therefore only present here throughput results for the different approaches already described, but do not alter all other possible parameters.

Table 1 shows that the manual instrumentation yields a speedup of around 20%. The dynamic approaches are even more successful, with a speedup of around 30% and 35% to the baseline, respectively.

We identified the following factors contributing to the additional performance gained by the dynamic approach over the static one, despite the additional overhead for call indirection and statistics collection:

- With the dynamic approach, we instrument all locks in the program, not only the single, common cache_lock. This includes locks in linked shared libraries.
- We instrument all paths for those locks (e.g., also the store path in memcached).

Additionally eliding rare paths might contribute overproportionally to reduce abort rates, as transactions in rare paths might abort serveral transactions in common paths (especially with high thread counts).

- The additional statistics collected contribute only indirectly to the additional speedup by restricting elision to effective locks. Statistics collection replaces the manual work of identifying relevant points for elision and will keep overhead down caused by useless elision.

Statistics might also prevent some overhead on locks whose profiles are not temporally stable, for example, caused by workload changes over time. We don’t expect to see this effect for the synthetic workload created by memslap though.

The results reported in Table 1 are also roughly in line with the blocking times we saw with mutrace (cf. to Section 4.1) and indicate that lock elision may be a good technique to eliminate huge portions of overhead due to pessimistic locking in memcached.

5. Related work

AMD has proposed ASF \cite{21} as a best-effort hardware transactional memory (HTM).

Converting critical sections requiring mutual exclusion into parallel code can be achieved in multiple ways. The original speculative lock elision proposal \cite{19, 20} is a mechanism that does not change the CPU architecture but relies on complex prediction to transparently detect locks and critical sections. We suggest to offload the adaptation logic to a transparent software layer and thus do not need complex and inflexible hardware predictors.

Azul’s Java-specific hardware transactional memory component allows parallel execution of Java’s synchronized methods, but relies on a modified JIT compiler and custom hardware design \cite{12}.

Dice et al. use the transactional memory implementation of the Rock processor to implement transactional lock elision \cite{13}. They modify the C++ standard vector class to use Rock’s transactional memory primitives and suggest an implementation inside a Java virtual machine. Our approach does not require recompilation, but makes the benefits of

**Table 1.** Throughput results for four memcached setups.

<table>
<thead>
<tr>
<th>Setup</th>
<th>Throughput (transactions / s)</th>
<th>Improvement (to baseline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>430 470</td>
<td>0.0%</td>
</tr>
<tr>
<td>Manual instr.</td>
<td>524 430</td>
<td>21.8%</td>
</tr>
<tr>
<td>Dynamic instr.(^a)</td>
<td>559 250</td>
<td>29.9%</td>
</tr>
<tr>
<td>Dynamic instr.(^b)</td>
<td>576 675</td>
<td>34.0%</td>
</tr>
</tbody>
</table>

\(^a\) Conservative back-in strategy

\(^b\) Aggressive back-in strategy
lock elision available to all applications using the standard
pthread mutexes. According to [13], Rock would have dif-
ficulties with such an approach due to hardware limitations
in Rock’s TM to execute arbitrary binary code (divisions,
function calls, TLB misses, ...).

Transactional memory (TM) in general offers an alter-
native to lock-protected critical sections; however, several
problems complicate the conversion process and hinder
adoption.

Traditionally, software transactional memory (STM) pro-
vided a simple library-based interface that required pro-
mers to manually annotate transaction begin and end, and
wrap all memory accesses within the transaction. To re-
duce the tedious and error-prone manual effort, compilers
for transactional memory have been proposed by industry
[3,23] and academia [6, 17], but language semantics are still in
draft state [20, 21, 31].

The enhanced compilers provide atomic blocks and in-
sert appropriate calls and new instructions automatically.
Zyulkyarov et al. convert a lock-based Quake server to trans-
actional memory [35]. Despite the use of a TM-enabled
compiler, they require significant amounts of manual inspec-
tion of the source code.

We do not need compiler support and laborious software
conversion, but provide instantaneous performance gains for
existing critical section annotations and offer further perform-
ance improvements for programmer tuning effort.

Felber et al. discuss transactional annotation of binary
code through binary translation [17], but still rely on man-
ually annotated transaction boundaries and cause a 3x per-
formance degradation.

Usui et al. proposed the concept of adaptive locks in [34],
which combine traditional mutexes with STM-backed code
paths. From a user’s point of view adaptive locks look simi-
lar to our dynamic instrumentation. Behind the scenes, how-
ever, different techniques are employed. For adaptive locks,
two code paths are created for all critical sections, one for
either mode of operation. The authors use a full compiler
tool-chain for recompiling programs and for instrumenting
all memory accesses for the STM paths. We only need to
provide one additional library to the system that is integrated
by the dynamic linker at start time as our extension to ASF
can directly execute legacy code in transactions and, unlike
STM, provides strong isolation. Usui et al. also use a de-
tailed cost-benefit model to limit the overhead for their STM
mode. In our hardware-based implementation, overheads are
small for execution in transactional mode. Our policy there-
fore only uses a simple and low-overhead success predictor
but not expensive-to-determine, anticipated costs. In the fu-
ture, more complex workloads or less capable hardware im-
plementations may benefit from more elaborate statistics.

6. Conclusion

For this work we applied AMD’s Advanced Synchronization
Facility (ASF) proposal (a hardware transactional memory)
to the domain of lock elision. We extended ASF with the
speculation-by-default mode to allow better re-using of ex-
isting code and to work with a smaller and simpler toolchain.

We used a combination of software and hardware ap-
proaches for lock elisions. Software was used for detecting
the actual locks (by manual instrumentation and dynamic
instrumentation), for register snapshotting, for providing a
software back-up path, and for implementing the policy of
when to elide. Hardware, in the form of ASF, was used for
detecting actual data conflicts at runtime and rolling back
in case of conflict. We experimented with different software
approaches and found that a more complex software scheme
paid off in the form of higher transaction throughput for the
memcached workload.

More investigations with more use cases are obviously
needed (and underway) to get a broader understanding. This
paper is meant as an initial report on our approach. These
early results show interesting speedups in the area of 20–
35%.

Acknowledgements

The research leading to these results has received fund-
ing from the European Community’s Seventh Framework
Programme (FP7/2007-2013) under grant agreement N°
216852. We also thank our colleagues Michael Hohmuth
and Dave Christie for fruitful discussions and feedback and
the anonymous reviewers for their advice on the paper.

References

memcached/wiki/BinaryProtocolRevamped
/code.sixapart.com/svn/memcached/trunk/server/
doc/threads.txt
TransactionalMemory
Facility - Proposed Architectural Specification, 2.1 edition,
March 2009.
Charles E. Leiserson, and Sean Lie. Unbounded transactional
memory. In Proceedings of the Eleventh International Sym-
posium on High-Performance Computer Architecture, pages
[6] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst,
Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin
Nowack, Torvald Riegel, Pascal Felber, Patrick Martlier, and
Etienne Riviere. Evaluation of AMD’s advanced synchroniza-
tion facility within a complete transactional memory stack.
In Christine Morin and Gilles Muller, editors, EuroSys, pages
Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad
Calder, and Osvaldo Colavin. Unbounded page-based trans-


