Commit on overflow

Srđan Stipić, Adrià Armejach, Osman Unsal, Adrián Cristal, Mateo Valero
Barcelona Supercomputing Center, Spain
{srdjan.stipic, adria.armejach, osman.unsal, adrian.cristal, mateo.valero}@bsc.es

***Abstract*—Current commercial CPUs have hardware support for speculative lock elision (SLE). SLE tries to elide the lock by speculatively executing lock protected critical section. If the speculation fails, SLE acquires the lock and re-executes the critical section non-speculatively.**

Latest Intel CPUs implement SLE and hardware transactional memory (HTM) where SLE uses HTM transactions to speculatively execute critical sections. HTM only supports bounded size transactions where non-conflicting transactions execute until they overflow and abort. Bounded sized transactions impose the limit on the size of SLE protected critical sections. Even worse, the current SLE implementation execute large non-conflicting critical sections twice; first time, speculatively in a transaction, and second time, non-speculatively by acquiring the lock at the beginning of the critical section. Ideally, SLE should execute all non-conflicting critical sections exactly once.

This paper introduces a *commit on overflow* (COO) transaction abort policy which – instead of aborting – commits overflowed transaction and continues executing it. We show the usefulness of COO while executing large SLE protected critical sections. Also, we show that our COO implementation preserves atomicity of SLE protected critical sections.

I. Introduction

The contributions of this paper are:

- We propose a *commit on abort* (COO) transaction abort policy which – instead of aborting – commits the overflowed transaction and continues executing it.
- We show how to implement COO on top of current SLE implementation with minimal hardware changes.
- We show how our COO implementation preserves atomicity of SLE protected critical sections.

II. Motivation - The problem statement

Intel's best effort HTM a.k.a. TSX (Transactional Synchronization Extensions) implements HTM on top of the MESIF cache coherence protocol. TSX extends MESIF with two new cache-line states: transactionally-read and transactionally-written states. When transaction starts, the CPU snapshots the state of CPU's register file so that transaction can restart in the case of a transactional abort. During the execution of a transaction, all the memory reads and writes access the cache transactionally where the cache sets cache-line states to transactionally-read and transactionally-written state, respectively. When the transaction reaches the commit instruction, the transactions commits by clearing all the transactional states in the cache. A transaction can abort before reaching the commit instruction due to transactional conflict or capacity overflow[1]. The transactional conflict abort happens when some other core invalidates the transactionally modified cache-line. The capacity overflow abort happens when the current running transaction evicts the transactionally modified cache-line from its cache(s) (due to cache associativity limits or due to cache capacity limits).

```
 1 def lock_acquire(lock_addr, lock_value):
 2   this.lock_addr = lock_addr
 3   this.lock_value = lock_value
 4   tx_status = tx_begin();
 5   if tx_status == TX_STARTED:
 6    this.lock_value_orig = tx_read(lock_addr)
 7   else:
 8     // Transaction aborted. Reexecute
 9     // the critical section non-speculatively.
10     lock MEM[lock_addr] = this.lock_value
11
12 def lock_release(lock_addr, lock_value):
13   if in_tx():
14     // ``lock release''
15     if (this.lock_addr == lock_addr) &&
16        (this.lock_value_orig == lock_value):
17       tx_commit()
18   else:
19     MEM[lock_addr] = lock_value
20
21 def abort(tx_status):
22   // tx_rollback() restores the register file
23   tx_rollabck(tx_status)
24   // from this point on,
25   // the execution continues
26   // in the body of tx_begin()
```

Fig. 1: **SLE** lock acquire/release instructions and abort handler. In the case of a transactional conflict, the CPU implicitly calls the abort handler.

Speculative lock elision (SLE) uses HTM to elide locks and execute critical sections speculatively in a transction. In the case of a transaction abort, SLE executes critical section non-speculatively by acquiring the lock. Figure 1 shows the implementation of SLE `lock_acquire` and `lock_release` instructions. At the beginning of the critical section, SLE executes speculative lock instruction and starts a hardware transaction. At the end of the critical section, SLE releases the speculative lock and commits the transaction. The speculative lock instruction never writes the value of the lock to the memory (as if lock was acquired and released with the same value). If the speculative execution of the critical section fails (transaction aborts), the SLE restarts the critical section non-

[1]The transaction can abort for other reasons but they are not relevant for the discussion

speculatively. SLE acquires speculative lock non-speculatively by writing the value of the lock to the memory. After the lock acquisition, SLE executes critical section non-transactionally. Non-speculative SLE execution guaranties that the critical section executes atomically because SLE releases the lock at the end of the non-speculative critical section.

A. Suboptimal execution

If the critical section is larger than the maximum hardware transaction size, SLE performs worse than regular non-speculative critical section. Particularly, SLE ties to execute the large critical section speculatively, aborts, and re-executes the critical section non-speculatively.

The following example shows a function that finds the maximum value of the array inside of the critical section (the `lock` protects the critical section):

```
def max(array, lock) {
  curr_max = 0
  lock_acquire(lock, SPECULATIVE=True)
  for i = 0 .. array.size():
    if array[i] > curr_max:
      curr_max = array[i]
  lock_relase(lock)
}
```

In the previous example, the array size (`array.size()`) is bigger than the maximum hardware transaction (`critical_size`). This creates wasted execution, because the critical section tries to execute hardware transaction that overflows. The wasted speculative execution behaves like the following code:

```
 1 def max(array, lock) {
 2   curr_max = 0
 3
 4   // wasted execution
 5   tx_begin()
 6   for i = 0 .. critical_size:
 7     if array[i] > curr_max:
 8       curr_max = array[i]
 9   tx_abort();
10
11   // non-speculative critical section
12   lock_acquire(lock, SPECULATIVE=False)
13   for i = 0 .. array.size():
14     if array[i] > curr_max:
15       curr_max = array[i]
16   lock_release(lock)
```

The function `max` executes the for loop twice; first time, in the transactions that aborts, and second time, in the non-speculative critical section. This creates wasted execution (lines 5-9).

B. COO implementation

COO implements a commit-on-overflow abort policy by modifying the abort handler. Figure 2 shows the abort handler that implementations COO. When the transaction overflows (and before any cache-line gets evicted from the cache), the abort handler tries to acquire the lock by executing transactional write. Then, the abort handler tries to commit the transaction. If the transactional write or commit fail, the transaction aborts and SLE restarts critical section non-speculatively.

```
1 def abort(tx_status):
2   if tx_status == TX_OVERFLOW:
3     // acquire the lock
4     tx_write(this.lock_addr, this.lock_value)
5     tx_commit()
6     // from this point on, we hold the lock
7   else:
8     tx_rollabck(tx_status)
```

Fig. 2: **Modified abort instruction.** The abort handler implements commit-on-overflow abort policy. When transaction overflows, the abort handler tries to acquire the lock (by executing transactional write) an to commit the transaction. If any of transactional write and commit can fail, the transaction aborts and SLE restarts critical section non-speculatively.

C. COO discussion

At the beginning of SLE critical section, SLE executes `lock_acquire` instruction that starts a transaction. Then, SLE transactionally reads the value of the lock (Figure 1, line 6). The transactional read inserts the cache-line (containing the lock) to CPU's local cache. This ensures that currently running transaction aborts when some other critical sections update the same lock.

When a transaction overflows, the CPU executes the abort handler. At this moment, the cache-line containing the lock is still present in the cache because the CPU executes the abort handler before any cache-line gets evicted from the cache. In the abort handler, COO tries to acquire the lock (Figure 2), and if successful, it tries to commit the transaction. The transaction executing abort handler might abort only if the other caches invalidate transactionally modified cache-lines in a time frame between lock acquiring and transaction committing (Figure 2, lines 4-5).

COO preserves the correct execution of the SLE critical section because the critical section acquires the lock before committing an overflowed transaction. This has the same effect as if SLE acquired the lock at the beginning of the critical section and executed non-speculatively to the point where the overflow happened.

When executing previous `max()` function with COO, the critical section executes as follows:

```
 1 def max(array, mutex m) {
 2   curr_max = 0
 3
 4   tx_begin()
 5   for i = 0 .. critical_size:
 6     if array[i] > curr_max:
 7       curr_max = array[i]
 8
 9   lock_acquire(lock, SPECULATIVE=False)
10   tx_commit();
11
```

```
12   for i = critical_size .. array.size():
13    if array[i] > curr_max:
14      curr_max = array[i]
15
16   m.lock_release()
```

This example shows that COO eliminates the unnecessary re-execution of the critical section (lines 4-10).

[1]

REFERENCES

[1] W. J. Armstrong, C. S. Graham, S. M. Lambeth, D. F. Moertl, P. E. Movall, G. M. Nordstrom, and T. R. Sand, "Interrupt and message batching apparatus and method," Jul. 4 2000, uS Patent 6,085,277.