Efficient lock elision in TSO

Eduardo José Gómez-Hernández (Thesis Co-Directed by Alberto Ros and Stefanos Kaxiras)

Department of Computer Engineering, University of Murcia, Spain
Department of Information Technology, Uppsala University, Sweden

UPPSALA UNIVERSITET
UNIVERSIDAD DE MURCIA
European Research Council
Established by the European Commission

Introduction

Current processors include multiple cores enabling real parallelism using threads, but they also introduce the need for synchronization, critical sections, atomicity, race conditions, among others. Right now, many programs can run very fast, but they stop scaling because of at least one of the previous reasons. In this work, we are dealing with critical sections and atomicity.

Motivation

- Processors have mechanism to stall execution, squash incorrect instructions, and re-execute needed instructions
- Load-linked and store-conditional is the most similar thing we have to use this mechanism for atomics, it cancels the store of the value if any other processor has access/modified the data
- Works like Speculative Lock Elision (SLE) [2] and Hardware Transactional Memory (HTM) [1] exploit this idea, but with a huge performance penalty when a re-execution happens
- In HTM, re-executions appears to happen about 74% $\sim$ 99% of the time in nearly all the applications [3]

Locks / Critical Sections

- When running a parallel application, some parts cannot be run in parallel, they have to be serialized
- Using Amdahl's law (Equation 1), the $\mathbf{(1-p)}$ part will be the serialized code
- This serialization establish an upper limit for speedup
- The main reason for serialization is the update of some shared memory location
- When possible to use, an atomic instruction will reduce the stall time
- The main idea behind atomic instructions is that the read and write operations will be execute atomically

$$S = \frac{1}{(1-p) + \frac{p}{s}} \quad (1)$$

Equation 1: Amdahl's law

A typical pthreads parallel section looks like:

```
for (int i = (tid * n/threads);
     i < ((tid + 1) * n/threads); ++i) {
       int x = op(i);
       // Critical Section
       pthread_mutex_lock(&lock);
       if (max < x)
         max = x;
       sum += x;
       pthread_mutex_unlock(&lock);
     }
```

Removing the parallel part, this is the critical section isolated:

```
pthread_mutex_lock(&lock);
if (max < x)
 max = x;
sum += x;
pthread_mutex_unlock(&lock);
```

Our approaches

- **Flexible Atomic Instructions**
  The main problem of atomic instruction is their rigidity and limitation. At least in x86 only integer atomic instructions exist, and their operations are very limited. A set of flexible atomic instructions will help to cover more cases than original x86 ones.

  In the previous example, using x86 atomics is only possible for part of the code:

```
pthread_mutex_lock(&lock);
if (max < x)
 max = x;
pthread_mutex_unlock(&lock);
fetch_and_add(&sum, x);
```

  But, having flexible atomics that allow selecting which operation and condition:

```
atomic(less, assign, &max, x);
atomic(none, add, &sum, x);
```

  The problem of the previous two solutions is that `max` and `sum` are not updated at the same time, in this example is not a problem, but in others could be, therefore using multi-address atomics they are updated in the same atomic group:

```
atomic(less, assign, &max, x,
       none, add, &sum, x);
```

- **Hardware multi-address mutex lock**
  Here we are approaching multiple problems at the same time. First, a hardware mutex will help to reduce lock/unlock overhead. Second, multi-address locking is not a trivial task, may problems can appear (most of them deadlocking). This approach is able to be treated as a mutex just by locking a common address (structure pointer, a specific field, etc).

  Using these locks/unlocks to inform the processor which variables should be protected from reading and writing. In this way, the code returns to be very similar to the original one:

```
lock(max);
lock(sum);
if (max < x)
 max = x;
sum += x;
unlock(max);
unlock(sum);
```

Goal & On-going work

- It is possible to stall the processor if needed when accessing the memory
- These stalls are directed by the answers of the coherence protocol
- Manipulating the coherence protocol is possible to generate the lock/unlock mechanism
- If a load or store is targeting a locked address by another processor, it will be delayed until it is unlocked from the locker processor
- To increase performance when using mutexes, a good critical block division should be made (like a table lock vs a lock per entry)
- With our approaches where variable addresses are used, there is no need of block division, this is made automatically

References

[1] T. Harris, J. Larus, and R. Rajwar. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, Dec. 2010.

[2] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *34th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 294–305, Dec. 2001.

[3] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance. In *2013*, pages 19:1–19:11, Nov. 2013.

Mail: eduardojose.gomez@um.es