Efficient Lock Elision in TSO

Eduardo José Gómez-Hernández*†[1],
(Thesis Co-Directed by Alberto Ros*
and Stefanos Kaxiras†)

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

ABSTRACT

Nowadays, processors count with a huge amount of cores, and applications that were to scale beyond the sequential performance use parallelism. The main problem of parallelism is synchronization, critical sections, atomicity, race conditions, etc. Focusing on the critical sections environment, Speculative Lock Elision and Hardware Transactional Memory are two interesting mechanisms, but when a re-execution is needed, the performance penalty is very high. In this work, we want to show our current approaches to mitigate the cost critical sections: Flexible Atomic Instructions and Hardware multi-address mutex lock. This is still an on-going work but we think they have the potential to improve in a significant way the performance of critical sections.

KEYWORDS: Microarchitecture, lock elision, TSO, atomic instructions

1 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.

When entering in a speculative execution environment, or after a fault appears, the processor will stall the execution, cancel the incorrect instructions and re-execute the ones that are needed. The nearest thing that is currently available is load-linked and store-conditional. This methodology allows to cancel the store if the data has been modified/accessed by another processor.

Following the original idea of squashing and re-execute, but applied to critical sections, there are two well-known mechanisms: Speculative Lock Elision (SLE) [RG01] and Hardware Transactional Memory (HTM) [HLR10]. The main point of these ideas is the speculative execution of critical sections, and if there is a collision from any other processor all the critical section is re-executed from a non-speculative state. The main problem is the performance impact this methodology has, in HTM, these re-executions appears to happen about $74\% \sim 99\%$ of the time in nearly all the applications [YHLR13].

[1]E-mail: eduardojose.gomez@um.es

2 Locks / Critical Sections

When running a parallel application, some parts cannot be run in parallel, they have to be serialized (in instance $(1-p)$ part in Equation 1), establishing an upper limit for speedup. Mainly because a shared memory location will be updated or modified. To avoid invalid results there are two solutions, serialize the section (critical section) or use atomic instructions.

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

Equation 1: Amdahl's law

If the operation to apply to that memory location is simple enough that an atomic instruction exists, this should be used as it has much less overhead than serialize the section. The idea behind atomic instructions is that the read and write operation will be executed atomically, therefore there is no room to another thread to read and write a wrong value.

3 Our Approaches

3.1 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:

```
1 pthread_mutex_lock(&lock);
2 if (max < x)
3  max = x;
4 pthread_mutex_unlock(&lock);
5 fetch_and_add(&sum, x);
```

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

```
1 atomic(less, assign, &max, x);
2 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:

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

3.2 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:

```
1 lock(max);
2 lock(sum);
3 if (max < x)
4  max = x;
5 sum += x;
6 unlock(max);
7 unlock(sum);
```

4 Goal and On-going work

With the current mechanisms, it is possible to stall the processor if needed when accessing the memory, but besides that, they are directed by the answers of the coherence protocol. Therefore, 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. In this way, it is possible to avoid re-execution.

As this mechanism treat addresses in loads and stores, it is easier to execute critical sections concurrently. To increase performance when using mutexes, a good critical block division should be made (like a table lock vs a lock per entry). In this case, this is not needed because it will be made automatically.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134).

References

[HLR10] Tim Harris, James Larus, and Ravi Rajwar. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, December 2010.

[RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *34th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 294–305, December 2001.

[YHLR13] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance. In *2013*, pages 19:1–19:11, November 2013.