Exploring weaker atomic instructions for latch-free data structures

Shengliang Xu
slxu@cs.washington.edu

1. INTRODUCTION

Since the rise of multi-core architectures in the recent years, multi-threaded concurrent programming quickly become one of the critical programming techniques for further improving program performances.

The biggest problem in concurrent programming is synchronizing access to shared memories among different threads. There are generally two ways of synchronization.

- The first one is blocking through locks. Before accessing an area of shared memories, a thread must first aquire a lock. If the lock is currently held by another thread, the aquiring thread will yield the CPU. OS is responsible for awaking threads who are chosen for execution after the lock is free.
- The second one is non-blocking synchronization through special atomic instructions. The de facto atomic instruction is the Compare-and-Swap (CAS). Most commonly, CAS is used to implement a spin lock. When a thread wants to access a shared memory block, it first gets a spin lock. Other threads wait the lock to be free by repeatedly testing the spin lock.

Atomic instruction based non-blocking synchronization has the advantage of low system overhead comparing with blocking synchronizations. Recently, a lot of research efforts have been put into developing latch-free data structures. And also, many higher level programming languages add direct support to them through libraries , such as Java (in the java.util.concurrent.atomic package).

There are several different atomic instructions available in current CPU architectures, with different semantics. Table 1 lists the most popular, if not all of, atomic instructions. Among them, CAS has the strongest functionality because we are able to use CAS to replace all others. Most of research and engineering on latch free data structure and problem solving are based on CAS. For example, OpenJDK 9 uses only CAS to implement all its atomic variable functionalities.

Table 1: Atomic instructions provided by various platforms

	Compare-and-Swap	Swap	Test-and-Set	Fetch-and-Add
ARM	LL/SC	deprecated	no	no
POWER	LL/SC	no	no	no
SPARC	yes	deprecated	yes	no
x86	yes	yes	yes	yes

How CAS works is that (this is how X86 implemented it) given a memory word m_1, compare the value of it with another value c, if the two matches, replace the value of m_1 to a new value m_2. The problem of CAS is that it can fail. The typical usage of CAS is using a while loop such as the following simple code block:

```
while (CAS fail) { ... }
```

It basically means keeping doing CAS until it succeeds. This can result in poor performances if the number of threads concurrently doing CAS is big.

On the other hand, the weaker atomic instructions succeed every time. This difference may result in highly efficient algorithms for some concurrent problems. For example, for an atomic integer ID generator, Fetch-and-Add should be able to outperform Compare-and-Swap a lot in high contention situations. Recently, Morrison et. al. designed a new FIFO queue using Fetch-and-Add [1] which dramatically outperforms CAS based and blocking locking based FIFO queues in high contention situations.

2. MILESTONES

Week 1 2: Add weak atomic instruction support to OpenJDK 9. And test the performance

Week 3 4: Find a data structure that is suitable for implementation using weaker atomic instructions.

Week 5 : Implement the chosen data structure.

3. REFERENCES

[1] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. ACM.