Group presentation - Sets

Konrad Pozniak, Nick Mayerhofer

Technical University of Vienna

0925996, 0726179

July 1, 2015

Overview

1. Introduction
2. More detailed set description
3. Everything else we did
    - Timer Benchmarking
    - Debugging of the LFS
4. Evaluation
    - Throughput
    - Fairness

Our tasks

- Implement five different skjghkjgkjggets
  - Reference Set using a C++11 std::set
  - Fine grained locking Set
  - Optimistic synchronization Set
  - Lazy synchronization Set
  - Lockfree Set
- find/implement a benchmarking process
- evaluate the set performance

Set Interface

```
class AMPSet {
[...]
//adds an item to the set [...]
virtual bool add(long item) = 0;

//removes an item from the set [...]
virtual bool remove(long item) = 0;

//checks if an item is contained in a set [...]
virtual bool contains(long item) = 0;
};
```

Reference

- Used C++11 `std::set`
- synchronized each call to the object with a global `std::mutex`
- lock, since it is not thread safe

Basic information

- `std::set` is based on a binary search tree
- our implementations will be based on simple lists
- the difference is going to be interesting

Fine grained locking

- *no* global lock, but individual node locking
- ⇒ multiple threads can operate at the same time on different locations of the list
- deadlock free, because of the lock ordering
- linearization point if item in set is at the corresponding locking, otherwise at the parents node locking

Optimistic synchronization

- does not lock any nodes during search, but when its found
- locked are the found element and its predecessor
- Requires validation that the nodes are still in list
  - Q: What happens if that's not the case?
  - restart necessary

Lazy synchronization

- does not acquire locks for `contains` checks
- ability to flag a node as *removed*
- consequence: locally removed, but may still be linked

Lockfree

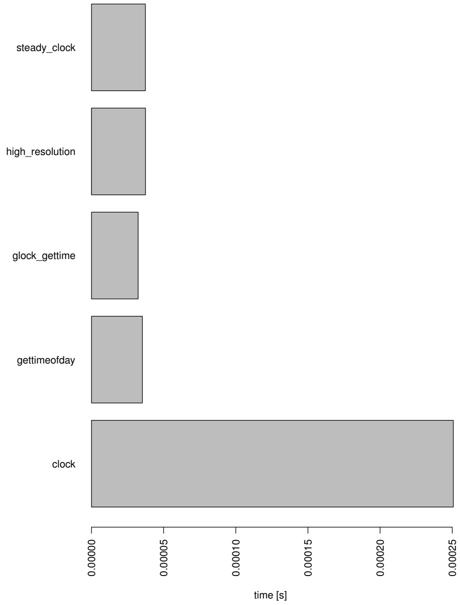
- no locks at all, but *hardware* atomic operations
- hardware support is provided due to combining pointer and flag into an atomic unit
- AMD64 .. 48bit with 64bit alignment
- SPARC T5 .. Physical 48bit (T4 44bit)
- tricky to implement

Timer benchmark

Benchmarking the timer, we ran each time measurement 1000 times

```
// c++11 steady clock - <chrono>
std::chrono::steady_clock::now();

// c++11 high res clock - <chrono>
std::chrono::high_resolution_clock::now();

// monotonic clock - <include/time.h>
clock_gettime(CLOCK_MONOTONIC, &tmpTimeNow);

// get time of day - <sys/time.h>
gettimeofday(&start , NULL);

// system clock - <include/time.h>
clock();
```

C++11 and Linux timer benchmarking, 649 datasets



We found bugs in the way of locking of the LFS

Node unlinked after mark

```
if(isMarked(w.curr)) {
    next = mark(next);
}
__sync_bool_compare_and_swap(
        &(getPointer(w.pred)->next),
        w.curr, next);
```

Node was removed just after 'find' found it unmarked

```
bool LockFreeSet::add(long item) {
  LfsNode *n = new LfsNode(item, nullptr);
  while (true) {
    LfsWindow w = find(item);
    if(isMarked(w.curr)) {
      continue;
    }
   [...]
  }
}
```

what did we analyze?

You are able to do a lot of benchmarks, a lot lot. We did the following:

- Performance comparison, with respect to throughput
  - between two machines [mars, ceres]
  - between four sets [REF, OS, LS, LF] (why *not* FGL)
  - between four operation types [insert, contains, remove, mixed]
- thread fairness comparison
  - between two machines [mars, ceres]
  - between five sets [REF, FGL, OS, LS, LF]
  - with just one operation type (why one)

Expectations

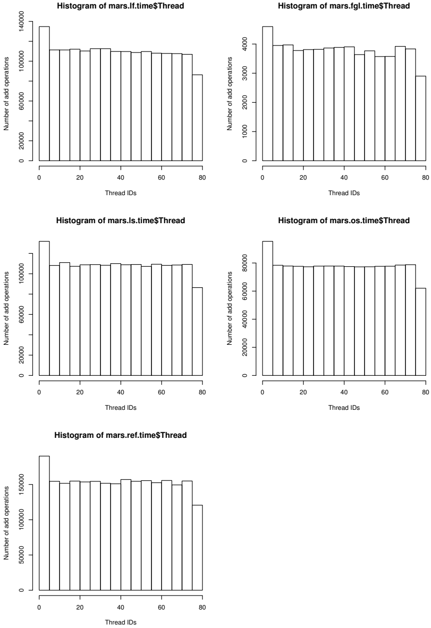
- a much faster reference set in single threaded mode
- at the beginning unknown expectations concerning parallel behavior

	add	contains	remove	mixed
reference	418.53	231.50	470.04	272.85
optimistic sync.	2609.88	418.71	3421.86	38.19
lazy sync.	1333.05	289.80	215.82	25.22
lock free	1128.28	161.50	115.61	29.39

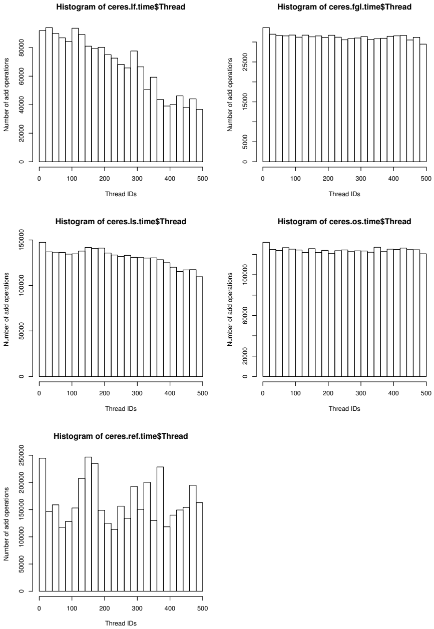
Average time in milliseconds of 100 throughput benchmark runs on Mars, 80 threads, 1000 iterations per thread

	add	contains	remove	mixed
reference	29.75	25.17	25.77	27.94
optimistic sync.	1348.86	634.34	2344.10	39.79
lazy sync.	635.12	328.49	307.21	30.92
lock free	687.51	320.21	358.25	16.03

Average time in milliseconds of 100 throughput benchmark runs on Ceres, 64 threads, 1000 iterations per thread



Histograms of 5 second runs on Mars with 80 threads



Histograms of 5 second runs on Ceres with 500 threads

Thank you