Improving the performance of Atomic Sections

Motivation

Processor manufacturers are no longer increasing clock speeds at the same rate they have done previously, due to the demands it places on power. Instead, they are now using increases in transistor density to put multiple processing cores on a chip. To harness this parallel computing power, software programs need to be multi-threaded.

However, shared memory multi-threaded programming is hard because threads can interfere with each other when they simultaneously read and write to the same memory location. Furthermore, such situations, also known as *race conditions*, can be extremely difficult to detect and debug because their occurrence depends on the non-deterministic way in which threads are interleaved.

Currently, programmers prevent such errors by ensuring that conflicting accesses are mutually exclusive, using locks. However lock-based programming is host to a number of difficulties itself, including:

- Locks are not always composable
- Locks can introduce deadlock
- Locks break modularity
- Not always possible to lock everything
- Priority inversion, convoying, starvation
- ...

Deadlock

T1:	T2:
Lock(A)	Lock(B)
Lock(B)	Lock(A)
Access A & B	Access A & B
Unlock(B)	Unlock(A)
Unlock(A)	Unlock(B)

T1	T2
Lock(A)	
	Lock(B)
Lock(B)	
Waiting	Lock(A)
Waiting	Waiting

Atomic Sections

This has led to the proposal of a high-level language construct called an *atomic section* that allows programmers to denote the parts of their program that should execute free of thread interferences but delegate the responsibility of enforcing this to the compiler/run-time. Semantically, the result of executing multiple atomic sections concurrently is the same as if they were run in some serial order. This is known as *serialisability:*

Serialisability

T1:
```
atomic {
   Access A & B
}
```

T2:
```
atomic {
   Access A & B
}
```

T1	T2
Access A & B	
	Access A & B

or

T1	T2
	Access A & B
Access A & B	

A naïve implementation of these semantics would be to acquire/release a global lock. However, this prevents non-conflicting atomic sections from executing in parallel. Hence, implementations strive to allow as much concurrency as possible while at the same time avoiding race conditions and deadlock.

The most popular technique for achieving this is *Transactional Memory*, which rolls back updates if a conflict is detected or, with current state-of-the-art implementations, if deadlock/starvation occurs. Although, this leads to the following problems:

- Irreversible operations, such as I/O and system calls, are hard
- Logging and roll-back can lead to high run-time overhead

Lock Inference

Another approach, called *lock inference*, is to statically infer which locks need to be acquired for atomicity and automatically insert lock()/unlock() in a way that deadlock is avoided.

How It Works

T1, T2:
```
atomic {
   Access A & B
}
```

Apply analysis →

T1, T2:
```
Lock(A)
Lock(B)
Access A & B
Unlock(B)
Unlock(A)
```

T1	T2
Lock(A)	
Lock(B)	Lock(A)
	Waiting
Unlock(B)	Waiting
Unlock(A)	Lock(B)

No deadlock!

The advantages of this approach are:

- It supports irreversible operations
- It provides excellent performance in the common case of no contention
- It has significantly less run-time overhead

In general, the objectives of lock inference are to:

- Maximise concurrency (use fine-grained locks and minimise time spent holding locks)
- Ensure freedom from deadlock (order locks or acquire all locks together)
- Minimise the number of locks (use multiple granularities of locks)

However, lock inference relies heavily on static analysis. Consequently, it has to be conservative about what to lock. The following make lock inference hard:

- Locks cannot be acquired after a lock has been released (two-phase locking)
- Aliasing
- Loops/recursion
- Large libraries
- Reflection
- Native method calls

Two-phase locking

```
Lock(A)
Unlock(A)
Lock(B)
```
Incorrect

Loops - e.g. linked lists

```
// What do we lock here?
while (n != null)
   n = n.next;
```

Current Work

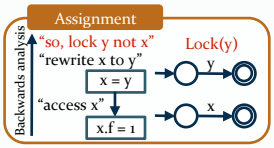
We are currently looking at how to infer fine-grained locks. The granularity of locking is tightly coupled with the granularity of the compile-time representation of objects. Some previous work represents objects by allocation sites. However, this can be coarse, as all objects constructed by the same statement are protected by the same lock.

Programmers refer to objects in their source code using lvalues (e.g. x.f). These lvalues can refer to any number of objects at run-time. Furthermore, in languages such as Java where each object is protected by its own lock, the lvalue not only gives us the run-time object being accessed but also the unique lock that protects it. Hence, we infer lvalues.

Our analysis uses finite state automata to precisely represent lvalues that may even contain unbounded accesses [1]. Furthermore, we can handle aliasing and assignments.

To support such fine-grained locking, we dynamically detect deadlock at run-time. However, so that no memory updates are performed by threads involved in a deadlock, (because we cannot rollback), we acquire all locks together at the start of an atomic section.

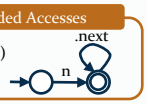
Assignment



We are currently scaling our approach to Java and are implementing it in the Soot framework. In order to support large programs, we have reformulated our analysis to be summary-based (IDE analysis).

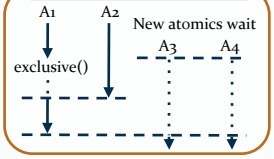
Unbounded Accesses

```
while (n != null)
   n = n.next;
```



The main challenge we are currently facing is imprecision introduced when using the class library. For example, *PrintStream.println(String)* has 1100 methods reachable from it in its static call graph, however, only a fraction of these methods get executed at run-time. Most of these additional methods are due to corner cases.

Exclusivity for precision



To solve this, we are considering a dynamic technique called "exclusivity," whereby our analysis ignores code paths that rarely get executed. However, to maintain atomicity when these paths are ever run, the current atomic section executes in "exclusive" mode. This means waiting for all other concurrent atomic sections to finish executing, then resuming execution till the end disallowing new atomic sections from starting. After the exclusive atomic has finished, new atomics can run again. Such code paths are rare and thus atomic sections will only rarely execute exclusively. We can also use this technique for dealing with reflection and unsafe native calls.

Exciting Future Work

1. Reducing the granularity of locks for unbounded and array accesses. For unbounded accesses, we consider using ownership types and for array accesses, locking the subregions of arrays that are accessed.

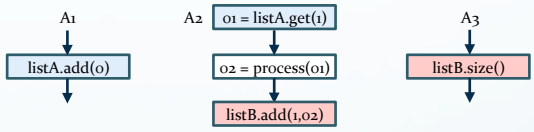
Loops - currently

```
Lock(Node class) // too coarse
while (n != null)
   n = n.next;
```

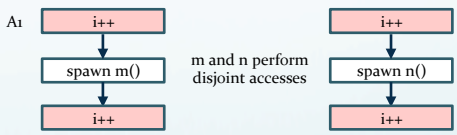
Arrays - currently

```
Lock(a) // too coarse
for (int i=0; i<a.length/2; i++)
   a[i] = 2;
```

2. Reduce the time spent holding locks by identifying the subregions of atomic sections that conflict with each other on a per-atomic basis and only serialising them. In the following example, the blue and red conflicts can be serialised independently:



3. Supporting parallelism inside an atomic section and using it to reduce the time spent holding a lock by forking off computations that can run in parallel.



4. Considering a hybrid implementation with transactional memory, giving the benefits of high concurrency from transactional memory, but reduced run-time overhead and more general support for irreversible operations from locks.

Publications

[1] David Cunningham, Khilan Gudka, and Susan Eisenbach. *Keep off the grass: Locking the right path for atomicity*. In L. J. Hendren, editor, CC, volume 4959 of Lecture Notes in Computer Science, pages 276–290. Springer, 2008.

Imperial College London

Khilan Gudka

Supervisors: Prof. Susan Eisenbach and Tim Harris

More info: http://www.doc.ic.ac.uk/~khilan/

Microsoft Research