Bibliography management: `thebibliography` environment

ShareLaTeX

September 21, 2017

Background Knowledge

0.1 Cuncurrent programming

Concurrent computing is a form of computing in which several computations are executed during overlapping time periodsconcurrentlyinstead of sequentially (one completing before the next starts). This is a property of a systemthis may be an individual program, a computer, or a networkand there is a separate execution point or "thread of control" for each computation ("process"). A concurrent system is one where a computation can advance without waiting for all other computations to complete. The main challenge in designing concurrent programs is concurrency control: ensuring the correct sequencing of the interactions or communications between different computational executions, and coordinating access to resources that are shared among executions. Potential problems include race conditions, deadlocks, and resource starvation. The scheduler is usually responsible for running a thread. Due to this scheduling indeterminism the programmer can not always be aware of which thread will be scheduled next and thus concurrent programms may seem to run randomly.

An important aspect of a concurrent program is the notion of the set of interleavings. If we imagine a process as a (possibly infinite) sequence/trace of statements (e.g. obtained by loop unfolding), then the set of possible interleavings of several processes consists of all possible sequences of statements of any of those process.

As it can be inferred, debugging of this kind of prgrams can be proved extremely challenging. The challenge mainly emerges from the fact that it is not always clear a thread command will be executed. Moreover the error may not always occur or cannot be traced during debugging. (bugs)

0.2 Stateless model checking and partial order reduction

In order to find an error on a concurrent algorithm one must examine every possible interleaving that this algorithm can produce. Usually the error would occur only under an unexpected interleaving, making its detection extremely difficult. Stateless model checking is based on the idea of driving the program along all possible interleavings. However this approach suffers from state explosion, i.e the number of all possible interleavings grows exponentially with the size of the program and the number of threads. In order to deal with this challenge

to approches have been proposed: partial order reduction and bounded search. Partial order reduction is aiming to reduce the number of interleavings explored by eliminating equivalent interleavings. Independent events do not change the program behavior. There are two ways that an partial order reduction algorithm can be implemented. The first is a static partial order reduction algorithm when the algorithm tracks the dependences between two threads before the execution of the concurrent program. The second is the Dynamic partial order reduction (DPOR) which observes the program's dependences on runtime. It is important to notice that the size of the state space still grows exponentially even if it is reducted by the DPOR.

For larger programs DPOR often runs longer thatn develpers are willing to wait. In these cases, bounded search can be proved usefull. Bounded search, in contrast to the DPOR, alleviates state-space explosion by pruning the executions that exceed a bound. [] There have been proposed many bounded techniques such as preemption bounded search or delay bounded search. The bounded search techniques are based on the notion that many of the concurrecny bugs can be tracked even when the bound limit is set to be small, thus the time required for a bug to be found is signifacantly smaller.

0.3 Vector Clocks

A vector clock is an algorithm for generating a partial ordering of events in a distributed or concurrent system and detecting causality violations. Just as in Lamport timestamps, interprocess messages contain the state of the sending process's logical clock. A vector clock of a system of N processes is an array/vector of N logical clocks, one clock per process; a local "smallest possible values" copy of the global clock-array is kept in each process, with the following rules for clock updates:

The algorithm follows the following specific steps:

1. Initially all clocks are zero.
2. Each a procss experiences an internal event, it increments its own logical clock in the vector by one.
3. Each time a process receives a message or performs an action on a shared variable, it increments its own logical clock in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message or the maximum value of all processes that share the same shared variable. (for every element).

Every algorithm that is presented in this thesis is based on vector clocks algorithm.

0.4 Unbounded dynamic partial order reduction

Many techniques have been proposed in order to implement a DPOR algorithm. What all these techniques share in common is this basic algorithm:

`Algorithm 1 DPOR`

```
Initially, Explore()
procedure Explore(S) begin
T = Sufficient set(final(S))
for all (t  T ) do
Explore(S.t)
```

A set of transitions is sufficient in a state s if any relevant state reachable via an enabled transition from s is also reach- able from s via at least one of the transitions in the sufficient set. A search can thus explore only the transitions in the sufficient set from s because all relevant states still remain reachable. The set containing all enabled threads is trivially sufficient in s, but smaller sufficient sets enable more state space reduction.

The algorithm above describes just a DFS search in the state space of all possible interleavings. As it be inferred from the algorithm the most important step is that of the calculation of the set T.

In bibliography many types of sets can be found[]. In this thesis we mainly focus on persistent sets and on source sets.

Persistent sets: A persistent set in a state s is a sufficient set of transitions to explore from s while maintaining local state reachability for acyclic state spaces [9]. A selective search using persistent sets explores a persistent set of transitions from each state s where $enabled(s) \neq$ and prunes enabled transitions that are not persistent in s. Godefroid defines a persistent set as follows.

A set T T of transitions enabled in a state s is persistent in s iff for all nonempty sequences of transitions from s in A G such that i dom() : i 6 T and for all t T , t last().

Intuitively the above defintion can be described as followed: If a $T \in T$ and there is another thread T' that can be executed until a command which is in a race with the T then T' belongs in the persistent set.

Source sets: D EFINITION 4.1 (Source Sets). Let E be an execution sequence, and let W be a set of sequences, such that E.w is an execution sequence for each w W . A set P of processes is a source set for W after E if for each w W we have WI [E] (w) P 6 = . u t

Intuitively a

0.5 Sleep sets

Another technique complementary to the persistent or source sets aiming to reduce the number of interleavings is the sleep set technique. Sleep sets prohibit visited transitions from executing again until the search explores a dependent transition. Assume that the search explores transition t from state s, backtracks t, then explores t 0 from s instead. Unless the search explores a transition that is dependent with t, no states are reachable via t 0 that were not already reachable via t from s. Thus, t sleeps unless a dependent transition is explored.

0.6 Comparing Persistent sets with Source Sets

It can be easily seen that the definition of source sets is much more relaxed than the definition of the persistent sets. This relaxation enables the source sets to be much more efficient than the persistent sets.

Algorithm comparison:

From the example above, it is clear the reason why source sets are an improvement over persistent sets is the fact that minimum?? source sets can eliminate sleep set blocked traces i.e. traces that would eventually be blocked by the sleep sets. An algorithm that would only calculate minimal source sets would be optimal, hence would never explore two equivalent interleavings.

0.7 Bounded search - preemption bounded search

Bounded search explores only executions that do not exceed a bound [5, 12, 14]. The bound may be any property of a sequence of transitions. A bound evaluation function Bv(S) computes the bounded value for a sequence of transitions S. A bound evaluation function Bv and bound c are inputs to bounded search. Bounded search may not visit all relevant reachable states; it visits only those that are reachable within the bound. If a search explores all relevant states reachable within the bound, then it provides bounded coverage.

Preemption-bounded search limits the number of preemp- tive context switches that occur in an execution [12]. The preemption bound is defined recursively as follows.

An algorithm that could describe a bounded search would be the following:

```
Algorithm 1 DPOR
Initially, Explore()
procedure Explore(S) begin
T = Sufficient set(final(S))
for all (t  T ) do
if (Bv(S.t)  c) then
Explore(S.t)
```

The only difference between the unbounded and the bounded version of the algorithm is the if statement which allows for an interleaving to be explored only if the bound has not been exceeded.

Definition 2.5. Preemption bound [13]. Pb(t) = 0 Pb(S.t) = Pb(S) + 1 if t.tid 6 = last(S).tid and otherwise Pb(S) last(S).tid enabled(final(S)

0.8 Preemption-bound persistent sets

A set that has been proposed as a sufficient for preemption bounded search is the preemption bounded persistent set.

A set T T of transitions enabled in a state s = final(S) is preemption-bound persistent in s iff for all nonempty sequences of transitions from s in A G(Pb,c) such that i dom(), i 6 T and for all t T , 1. Pb(S.t) Pb(S. 1) 2. if Pb(S.t) ¡ Pb(S. 1), then t last() and t next(final(S.), last().tid) 3. if Pb(S.t) = Pb(S. 1), then ext(s, t) last() and ext(s, t) next(final(S.), last().tid)

Let us assume that P is a persistent set. A preemption bounded persistent set is a set that contains all $p \in P$ with the addition of all the threads that would be added in a block that would be created when a p was scheduled. These threads are called conservative threads threads and their goal is to allow the coverage of interleavings that would not exceed the bound. Notice that an interleaving can be both conservative and non-conservative.

1 Nidhugg

Nidhugg is a bug-finding tool which targets bugs caused by concurrency and relaxed memory consistency in concurrent programs. It works on the level of LLVM internal representation, which means that it can be used for programs written in languages such as C or C++.

By the time this thesis was written Nidhugg had been supporting the SC, TSO, PSO, POWER and ARM memory models. Target programs should use pthreads for concurrency, and each thread should be deterministic when run in isolation.

1.1 The Nidhugg's algorithm - Source-DPOR algorithm

Nidhugg implements the Source-DPOR algorithm[].

.

```
Initially Explore(hi, );
Explore(E, Sleep) ;
if p  (enabled(s [E] ) \ Sleep) then
backtrack(E) := {p} ;
while p  (backtrack(E) \ Sleep) do
foreach e  dom(E) such that(e - E.p next [E] (p))
do
let E 0 = pre(E, e);
let v = notdep(e, E).p ;
if I [E 0 ] (v)  backtrack(E 0 ) =  then
add some q 0  I [E 0 ] (v) to backtrack(E 0 );
let Sleep 0 := {q  Sleep | E|=pq} ;
Explore(E.p, Sleep 0 );
add p to Sleep ;
```

2 Implementation Details

2.1 Source sets

2.2 Bound count

2.3 Persistent sets

2.4 BPOR

2.5 Source-based BPOR

3 Evaluation

3.1 Applications on rcu

3.2 Why source-sets fail

4 Related work

5 Future work

This document is an example of `thebibliography` environment using in bibliography management. Three items are cited: *The LaTeX Companion* book [1], the Einstein journal paper [2], and the Donald Knuth's website [3]. The LaTeX related items are [1, 3].

References

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.

[2] Albert Einstein. *Zur Elektrodynamik bewegter Körper*. (German) [*On the electrodynamics of moving bodies*]. Annalen der Physik, 322(10):891921, 1905.

[3] Knuth: Computers and Typesetting,
`http://www-cs-faculty.stanford.edu/~uno/abcde.html`