Term Paper for IISc. Summer Course

Abhik Roychoudhury
Department of Computer Science
National University of Singapore

Overall Organization

- The setting of the overall problem
  - Memory Model sensitive Software Verif.
  - What is a memory model and why it is important?
- We will discuss a solution for the problem – in the context of C# program verification.
- **Your term paper should explore building a mem. Model sensitive checker for Java**
  - **Java Mem. Model is much more complicated** ☹
  - Prelim. solutions developed are also OK, you do not need to come up with a complete soln./impl. ☺

An example

Initially : A = 0, flag = 0

A = 1;		while (flag == 0) {};
flag = 1;	||	print A;

Expected value = 1

Possible printed values {0, 1}

The two threads are executed on different processors. The processors allow out-of-order execution (as long as data dependencies are not violated).

Sequential Consistency

Programmer expects statements within a thread to complete in program order: **Sequential Consistency**

1. **Each thread proceeds in program order**
2. **Operations across threads are interleaved**
   Op1 Op'' Op2 Op' violates SC

Op1;	||	Op' ;
Op2;		Op'' ;

Is this a problem ?

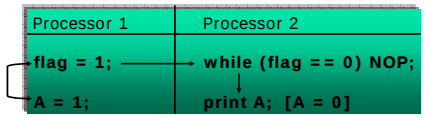
- Programmers expect SC
- Verification techniques assume SC
  - All existing model checkers assume SC
- Not demanded by C# lang. spec.
  - Violation of SC on real platforms leads to unexpected results in "verified" programs
  - "**C# Memory Model**" weaker than SC
- YES !!

Hardware memory model

- Defines the values that can be returned by a shared variable read operation
- Constrains the reordering of memory accesses to same/different locations
- Eliminates the gap between programmer's expectation and actual system behavior

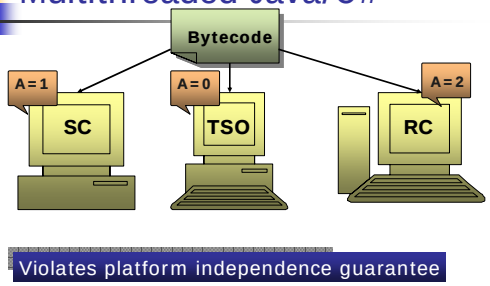
Relaxed HW memory models

- SC disallows compiler/hw optimizations.
- Relaxed memory models allow different level of memory operation reorderings
  - Never violate program dependencies



- Examples: TSO < PSO < WO < RC

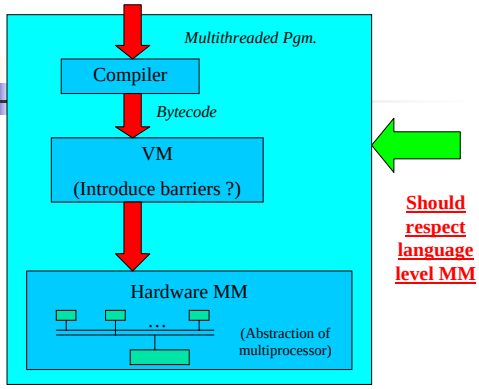
Multithreaded Java/C#



Violates platform independence guarantee

Language level Memory Model

- Software memory consistency model
- Defines the set of all possible behaviors of a multithreaded Java/C# program on any implementation platform
- Provides platform independence for multithreaded Java/C# programs on shared memory multiprocessors
- More relaxed than SC to provide compiler/ hardware optimizations opportunities



What is the impact on verification

- Programmers expect SC
  - Program verifiers use SC
- Platform provides a weaker guarantee
  - C# Memory Model
  - Lot of work possible in designing MM, impact of candidate MM on multiproc. performance!
    - How to enforce language level MM on platforms
- MM sensitive program verification
  - Language level MM contract between programmer and compiler/hardware designers.

Could we avoid the problem...

- ... instead of solving it? Try pgm modifications which can be automated
  - Re-ordering of operations within a thread should not be visible to other threads.
  - Threads communicate via shared variables.
    - Explicitly insert memory barriers between shared variable accesses, or indirectly
    - Enclose shared variable accesses with locks which typically flush out incomplete operations.

Memory barriers

- How to avoid a particular re-ordering
- Barriers between two operations enforce program order

Processor 1	Processor 2
A = 1; BARRIER (W↑) flag = 1;	while (flag == 0) NOP; print A; [A = 1]

- Non-intuitive to manually insert these, unless you indiscriminately introduce barriers for each shared var. access.

Is such a simple solution feasible?

- NO !!
  - Inserting memory barriers between all shared ops. causes big loss of performance.
    - Also barrier insertion should be done automatically
  - Synchronizing all shared variable accesses is done by many programmers, but
    - Programmers may avoid synch. overhead
    - May forget to synchronize (Common problem)

Another solution which does not work

- Languages like C# allow variables to be marked as volatile
  - Accesses to a volatile variable accesses its "master copy", but overheads lower than synchronization.
    - Vol rd/wr claimed to have "lock acquire/release semantics"
  - So, why not mark all shared variables as volatile ?
    - You can try this as a light-weight approach in your term paper, but not the only one hopefully!

Overall Organization

- The setting of the overall problem
  - Memory Model sensitive Software Verif.
  - What is a memory model and why it is important?
- We will discuss a solution for the problem – in the context of C# program verification.
- **Your term paper explores building a mem. Model sensitive bytecode checker for Java**
  - **Java Mem. Model is much more complicated** ☹
  - Prelim. solutions developed are also OK, you do not need to come up with a complete soln./impl. ☺

Volatiles do not work

- C# language spec. allows
  - Write A → Read B re-orderings for vol. vars. A, B
  - Not exactly acquire-release semantics !!
- C# implementations (.NET 2.0) allow such re-orderings, confirmed by experiments.
  - More on this, later.

Approach (1)

- Formally specify bytecode re-orderings within a thread allowed by (informal) C# MM spec.
  - Mem. Model may change later --- which opt. allowed?
    - Effect of compiler opt. reflected already in bytecode.
  - Should the MM spec. be executable ?
- Caution: The approach taken for MM spec. in C# may not be applicable for Java (why?)

Approach (2)

- Use the formal MM spec. to build a bytecode checker for invariant properties
  - Error detection: non-SC counter-example traces not traversed by traditional model checkers.
  - Error Correction: Use minimal # of barriers to remove all non-SC counterexample traces.

Bytecode re-orderings

Reorder	2nd byte code					
1st bytecode	Read	Write	Volatile Read	Volatile Write	Lock	Unlock
Read	Yes	Yes	Yes	No	Yes	No
Write	Yes	Yes	Yes	No	Yes	No
Volatile Read	No	No	No	No	No	No
Volatile Write	Yes	Yes	Yes	No	Yes	No
Lock	No	No	No	No	No	No
Unlock	Yes	Yes	Yes	No	No	No

Op1: X = 1

Op2: read Y X,Y are marked as volatile variables

Op2 may be completed before Op1

Such re-orderings do matter

Initially lock0 = 0; lock1=0; turn=0

```
lock0=1                          lock1=1
turn=1                           turn=0
while(1){                        while(1){
  if(lock1!=1)or(turn==0)          if(lock0!=1)or(turn==1)
    break;                           break;
}                                }
counter++;                       counter++;
lock0=0                          lock1=0
```

Mutual exclusion of Peterson's ensures counter = 2 at end of execution.

On .NET 2.0 even with all shared vars. as volatile yields counter =1

Bytecode re-orderings

- Divide bytecodes into types (based on opcode)
  - Read, Write, Volatile-read, V-Write, Lock, Unlock
- Simply define a matrix which captures the pair of re-orderings allowed
  - Non-executable specification which needs to be consulted by model checker during search.
  - More accessible to non FM-ers, but still
  - Completely Formal.

Bytecode level invariant checker

- After specifying re-orderings, things to do ---
  - Managing out-of-order exec during traversal
    - Role of re-ordering matrix
  - Explicit-state Bytecode level MC
    - Architecture similar to JPF for Java
  - Fixing the non-SC counterexamples produced
    - Using mincut algorithm on state transition graph.

Managing out-of-order exec

- Manage this within a model checker.
  - Split execution of a bytecode into issuing and completing stages
    - Issuing bytecode in-order
    - Completing bytecode may not be in-order
    - Maintain a list of incomplete bytecodes
    - Use re-ordering matrix to check which incomplete bytecode can be completed at the current state.

Example

- Write a; Read b;
  - Possible Execution
    - Schedule write to a
      - Incomplete actions: [wr a]
    - Schedule read of b
      - Incomplete actions: [wr a, rd b]
  - Can complete rd now, if wr -> rd re-ordering OK.

States

- One state consists of
  - Image of the heap
  - Threads' local variables, stacks
  - Threads' program counter
  - Threads' incomplete actions
- Each transition is either issuing a bytecode or complete an incomplete one

Transitions

- Executing a bytecode
  - Completes immediately
    - (branching/arithmetic/comparison/synchronization)
  - Schedule an *incomplete action*
    - (read/write/lock/unlock)
  - Complete an *incomplete action*
- **Explicit-state bytecode level reachability analysis**

(Lot of) Search Optimizations

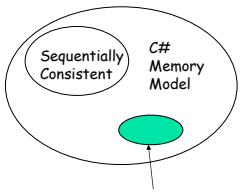
- Avoid re-ordering of thread local transitions within a thread, for example.
- What is thread-local ?
  - The transition operates on data not accessible by other threads
    - Local data of a thread
    - Heap data only reachable by the thread
      - Use program analysis methods – "Escape Analysis"

Using the checker

- Given an inv. (should hold for all reachable states)
  - Find all violating states reached under SC (without re-orderings)
    - Suppose, none.
  - Find all violating states reached using re-orderings
    - Suppose, VS = { s1, ..., sk}
  - Disable minimal # of transitions to prevent executions from reaching VS

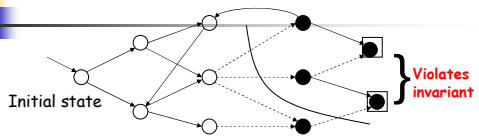
From error detection to correction

- C# memory model allows more states than S.C.
- Only a subset of states violate a property
- Enforcing S.C. is too inefficient
- We should disallow the trace to go to the shaded states only.



Set of states that violate the user's invariant property

Disabling unwanted states



- White states are only reachable under sequential consistency.
- Black states can be reached under C#/.NET memory model.
- Disable minimal re-orderings to avoid reaching violating states.
- Achieved by maxflow-mincut algorithm
  - Weight of solid edges = infinity
  - Weight of dashed edges = 1

Sample Programs

Benchmark	Description	# bytecode
peterson	Peterson's Mutual exclusion algorithm	120
tbarrier	Tournament barrier algorithm	153
dc	Double-checked locking pattern	77
rw-vol	Read-after-Write Java volatile semantic test	92
rowo	Multiprocessor diagnostic tests ARCHTEST (ROWO)	87
po	Multiprocessor diagnostic tests ARCHTEST (PO)	132
iw1	Independent workers problem 1	102
iw2	Independent workers problem 2	105
rw	Two readers-single writer lock algorithm	161
bb	Bounded buffer/Producer-consumer problem	252

Experimental results

	Normal			With POR			
Benchmar	#states	#transitions	time(s)	#states	#transitions	time(s)	#Barriers
peterson	903	2794	0.91	881	2697	0.9	3
tbarrier	1579	5812	1.51	801	2641	1.5	3
dc	228	479	0.31	150	272	0.36	1
rw-vol	1646	5616	2.02	1613	5458	2.06	4
rowo	1831	4413	1.41	1776	4231	1.36	2
po	6143	22875	7.71	6091	22595	7.64	6
iw1	14886	67589	22.81	246	557	0.59	0
iw2	92613	285430	86.53	353	560	0.63	0
rw	7762	28310	8.82	6851	22119	7.48	0
bb	16072	51447	**25.11**	1704	4048	**2.08**	0

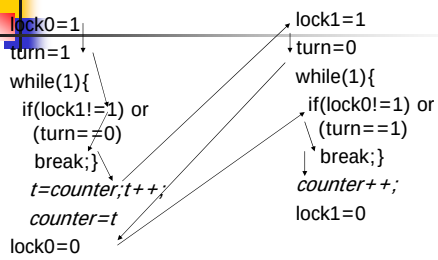
Going back and re-capitulating

Initially lock0 == 0; lock1==0; turn==0

```
lock0=1                    lock1=1
turn=1                     turn=0
while(1){                  while(1){
  if(lock1!=1) or (turn==0)   if(lock0!=1) or (turn==1)
    break;}                    break;}
counter++;                 counter++;
lock0=0                    lock1=0
```

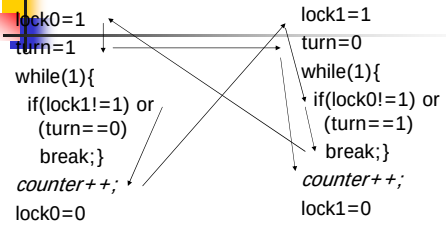
Peterson's Mutual Exclusion Algorithm

Initially lock0 == 0; lock1==0; turn==0



Broken trace under Programming Language Level Memory Model (C#)

Initially lock0 = 0; lock1=0; turn=0



Fix with volatile variables does not work (assume all shared vars. as volatile)

We need to build checkers!

Summary

- Memory Models for Prog. Languages
  - Traditionally not considered in MC
    - Capture re-orderings at bytecode level.
  - Specifying them as a re-ordering matrix
    - Look-up matrix during explicit-state MC
  - Use MC to find "violating" states which would not be reached under SC
    - Repair concurrency bugs using mincut algorithm
- Bytecode level MM sensitive verification

References

- A Memory Model Sensitive Checker for C#, T.Q. Huynh and A. Roychoudhury, FM 2006.
  - http://www.comp.nus.edu.sg/~release/mmchecker
- Impact of Java Memory Model on Out-of-order Multiprocessors, T. Mitra, A. Roychoudhury, Q. Shen, PACT 2004.
- Specifying Multithreaded Java Semantics for Program Verification, A. Roychoudhury and T. Mitra, ICSE 2002.
- Reasoning about Hardware and Software Memory Models A. Roychoudhury, ICFEM 2002.

And, now it is up to you !!

- The setting of the overall problem
  - Memory Model sensitive Software Verif.
  - What is a memory model and why it is important?
- We will discuss a solution for the problem – in the context of C# program verification.
- **Your term paper explores building a mem. Model sensitive bytecode checker for Java**
  - **Java Mem. Model is more complicated** ☹
  - Prelim. solutions developed are also OK, you do not need to come up with a complete soln.☺

Materials to start with

- To understand the Java memory model in full you can see http://www.cs.umd.edu/users/jmanson/java/journal.pdf
  - **The POPL 2005 conference version is better to start with than the journal paper though.**
- Important diff. with C# memory model
  - C# MM does not allow re-ordering which violate data dependencies within a program.
  - Even such re-orderings are allowed in the JMM!
  - Bigger search space to be explored by JMM sensitive checker

Materials to start with

The Java memory model (JMM) might be rather hard to understand at one go. So, it is reasonable to start thinking of a model which is close enough to the JMM, and preserves most of the ideas in the current JMM.

Here is such a model you can start with. It is also cited in the POPL'05 conference paper.

http://rsim.cs.uiuc.edu/~sadve/jmm/

Materials to start with

- To understand memory model sensitive verification (but done at a smaller scale for C# programs), read http://www.comp.nus.edu.sg/~abhik/pdf/fm06.pdf
  - This is what we discussed today to give you a feel !
- Prototype tool from http://www.comp.nus.edu.sg/~release/mmchecker/
- Open-source tool
  - You may want to download it and try to understand the state space representation and/or exploration

Materials to start with

- The following paper is also useful for understanding the issues in bytecode level reasoning (specifically via the model checking technique). However, this paper does not consider memory model issues.
  http://www.inf.ethz.ch/personal/basin/pubs/bcvjar.pdf
- **GOOD LUCK TO ALL !**

Material not to start with

The JMM as stated in the journal paper provides the crux of the official semantics document from Sun Microsystems. If you want to look at the official document, see
http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf
But this official document contains a lot of details which might not be relevant for our project. So it is best not to start with this document.

In addition, the page
http://www.cs.umd.edu/~pugh/java/memoryModel/
also contains lots of other references on this topic, but much of this stuff is outdated. So better not spend too much time going through all the links in this page. **Instead start with the POPL 2005 paper, and the SC- memory model of Sarita Adve.**

Administrative Details.

- You get 5 weeks to work on it (25% of marks)
  - Due Monday June 25, 2007.
  - We will have presentations in class on that day.
  - You will also need to submit a detailed report, and the code for implementation (if any).
- Get started immediately, there will too many assignments running in parallel with the term paper !
- You may work in individually or groups of 2.
  - I prefer that you form groups early and e-mail me the groupings to abhik@csa.iisc.ernet.in by Fri May 25.