Runtime Monitoring

Outline

- Runtime monitoring of code
- Two topics
  - Detecting data races
  - Machine simulation

Data Races

- Data races are a multithreaded bug
  - At least two threads access a shared variable
  - At least one of the thread writes the variable
  - The accesses are (potentially) simultaneous
- Races are usually undesirable
  - Source of nondeterminism
    - Program state depends on timing
  - Bugs are very hard to identify or reproduce

Data Races (Cont.)

- Note: Not all data races are bad
  - Just the vast majority are bad
- Example
  - Threads execute

```
if (predicate) x = 1
```

  - Threads where test passes race to set `x`
    - But `x` will be `1` if any thread's test is true

Happens Before

- Event A *happens before* event B if
  - B follows A in a single thread of control
  - A in thread a, B in thread b, event c such that
    - A happens in a
    - c is a synch event after a in A and before b in B
    - B happens in b
- This is the natural partial order of events

Pre-Eraser

- First race detection tools based on the *happens before* relation
- Sketch
  - Monitor all data references and synch operations
  - Watch for
    - Accesses of v in thread 1
    - Accesses of v in thread 2
    - With no intervening synch between 1 and 2

Problems

- This is expensive
  - Requires per thread
    - List of accesses to shared data
    - List of synchronization operations
- False negatives
  - Can miss races
  - Needs to be tested with many schedules

Thread 1	Thread 2
`y = y + 1`	`lock(m)`
`lock(m)`	`unlock(m)`
`unlock(m)`	`y = y + 1`

A Different Approach

- Happens-before tools look for actual races
  - Moments in time when multiple threads access a shared variable without protection
- A different approach is to check invariants
  - Look for examples that violate invariants that might lead to races

The Discipline

- Shared variables are protected by locks
- Discipline:
  - Every access to a shared variable is protected by at least one lock
  - Any access to a shared variable unprotected by a lock is an error

Which Lock?

- How do we know which lock protects a variable?
  - The program may hold many unrelated locks
  - Linkage between locks and shared variables undeclared
- Issue
  - Like any instrumentation approach, we don't have the resources to do intensive analysis during execution

Locksets

- Idea 1: Infer the locks
- Observation: It must be one of the locks held at the time of access

```
Initialize C(v) to the set of all locks (for each v)
On access to v by thread t
   $C(v) \leftarrow C(v) \cap$ locks_held(t);
   if $C(v) = \varnothing$ then print(warning);
```

Problems

- This doesn't quite work
- We need to deal with
  - Uninitialized Data
  - Read-Shared Data
  - Read-Write Locks

Uninitialized Data

- Data often initialized by one owner
- No need to lock at this time
- How do we know when initialization is done?
  - Answer: We do not
  - But, we can tell when the value is accessed by a second thread

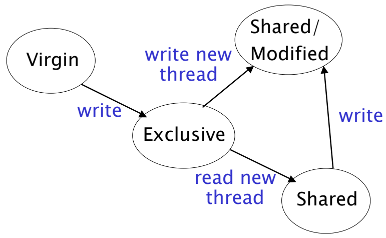
---

Read-Shared Data

- Once created, some data is only read
- No need to lock lock-free data
- Idea: Don't update locksets until at least
  - more than one thread has the value
  - at least one is writing to the value

State Transitions

- Each value (memory location) is in one of four states:



New Algorithm

- The new algorithm is as before
- But only locations in the shared-modified state have locksets inferred
- None of the other cases requires checking

Read-Write Locks

- Single writer, multiple reader locks
- Discipline: Some lock (a particular one) must be held in either write mode or read mode for all accesses of a shared location
- Locks can be held either in *write mode* or in *read mode*

Solution

- Refine computation of locksets to express single write exclusivety
- For each read of a location, compute

  $C(v) \leftarrow C(v) \cap \text{locks\_held}(t);$
- For each write of a location, compute

  $C(v) \leftarrow C(v) \cap \text{write\_locks\_held}(t);$

Implementation

- Done at the binary level
  - Could have been a source code tool
- Every memory word has a shadow word
  - 30 bits designated for the lockset key
    - Sets of locks represented by small integers in a hashtable
    - Depends on having not very many distinct sets of locks
  - 2 bits for state in the DFA

Results

- This works
  - Checking the discipline finds errors with few runs
  - Many imitators
- Eraser is slow
  - 10-30x slowdown
  - Could be made faster with static analysis
- Many Eraser-like tools available now

Opinion

- Runtime monitoring is a good idea
  - Especially at the “right-level”
  - Cf., program checking
- But it is painful to do
  - Binary instrumentation is a hassle
  - Mapping between source and binary is opaque
  - Performance is poor without a lot of effort

Machine Translation

- Idea:
  - Don’t run the program on the hardware
  - Do run the program on the virtual machine
- The ultimate in runtime monitoring
  - Full control of every instruction
  - A true universal machine
    - In the sense of Turing

Performance

- But virtual machines are slow
  - Surprise
  - More than 10x slowdown for naïve implementation
    - And much more for detailed simulations of e.g., caches
- Idea:
  - Use dynamic binary translation
  - Translate simulated code to native code
    - On the fly

Dynamic Translation

- Basic blocks are the unit of code translation
- Translated operations work on simulated state
  - Simulated machine registers stored in memory
  - Simulated PC tracked by code where needed

```
load r3, 16(r1)   ⇒   load t1 simRegs[1]
                      load t2 16(t1)
                      store t2 simRegs[3]
```

Translation Cache

- Translation is expensive
- Maintain a translation cache
  - Maps program counter ⇒ translated basic block
  - or calls translator if needed
- A detail
  - Must detect self-modifying code
  - Flush translation cache
  - Done by detecting writes to translated pages

Chaining

- Translated basic blocks end by jumping to main dispatch loop
  - Dispatch is on program counter
- *Chaining* is an optimization
  - Short-circuit path through dispatch loop if target of next basic block is statically known

Modeling the Memory Management Unit

- Embra’s goal is to simulate full workloads
  - Including the host OS
- This requires modeling virtual memory
  - In particular, the MMU
    - Mapping of virtual addresses to physical addresses
  - Because MMU operations are visible to the OS

MMU Relocation Array

- Maintain an array indexed by virtual page
  - Array size = memory size / page size
  - Array entries contain
    - Address of physical page for the virtual page
    - Protection bits
      - Valid/invalid, readable, writable

Dynamic Translation Revisited

- Each memory reference is translated to
  - Look up information in the MMU relocation array
  - Check the protection bits
  - Call out to exception routines if necessary
  - Construct physical address
- Requires 8 (optimized) instructions

A Performance Bound

- Memory operation requires 8 instructions
- Approx 1/3 of instructions are loads/stores
- Implies a minimum slowdown of 3x
  - Embra comes fairly close to this bound

Back to Dynamic Translation

- Modeling the MMU breaks chaining
  - Why? Because processes may have different code at same virtual address
  - But this is rare
- Solution
  - Use physical addresses for chaining
  - When executing translated block, first check that virtual PC and address of code agree
  - If not, go back through main dispatch loop

More Chaining

- Embro also does speculative chaining
- Chain any indirect jump
  - Presumably to the place it went last time
  - Before executing code, check it has correct virtual and physical address
  - A kind of caching
- Significant improvement
  - 20% on some benchmarks

Beyond the MMU

- Embra is designed to support ad hoc translations
- Example: Accurately simulating caches
  - Complete 2nd level cache simulation
  - Reported in paper

Other Neat Stuff

- Self-hosting studies
  - Embra simulating Embra
- Fast-forward studies
  - Try workloads on future machines
  - With more cache memory, MIPS
- Multiprocessor studies
  - On one processor
  - On real multiprocessors

What Happened?

- Embra became VMWare
- Very widely used because
  - Increases reliability by providing full isolation
  - Solves the 1 OS/1 machine problem
- This always was a good idea
  - Virtual machines were first pursued by IBM for the same reasons 30 years ago

Summary

- Runtime monitoring is useful
  - Debugging, performance analysis, safety, etc.
- Key is not to take too much time
  - In particular, no time for global analysis
  - Reasonable applications 10%-500% overhead
- Dynamic binary translation is the limit
  - Cheaper techniques approximate translation