How to Detect Self-Modifying Code During Instruction-Set Simulation

Pardo

AMAS-BT 2009

Background

- Helped invent & build two significant simulators
  - Shade
  - Crusoe
- Also: studied and written lots of SMC
  - Lots!
- Maybe: something interesting to say

Who Cares?

- Everybody knows

  Self-Modifying Code is dead

- But: SMC is alive and well (alive and still sick?)
  - Dynamic linking, JITs, debuggers, …
- Instruction Space Changes (ISC)
  - Demand paging - reuse code pages
  - Memory remapping
- ”SMC is everywhere”

Simulating SMC

- Rare: ”expensive” is okay
- Frequent: ”expensive” is slow
- Is slow okay? Depends on requirements
- How slow?
- Many forms of SMC: ”no silver bullet”
  - Code and writable data interleaved
  - Fine-grained JIT
  - Instruction patching
    - Immediates, opcodes, registers, ...

Strategies

- Easy solution: interpret everything
  - Decode every time → see every change
  - Slow...
- Fast and almost easy:
  - Translate, don't handle SMC/ISC
  - Many workloads won't run
- Fast and handle SMC/ISC:
  - Some cases: almost easy
  - General case: hard... but almost possible!

Simulator Structure

- Interpreter:

```
instr = fetch(pc)
h = decode(instr)
execute(h)
```

- Decode is slow, so cache:

```
h = cache.lookup(pc)
if (!h)
   h = cache.save(pc, decode(fetch(pc)))
execute(h)
```

- Avoids ”fetch” and ”decode” except on miss
- But:
  - What if the instruction changes?
  - What if the PC mapping changes?

Trap On Write

- Write-protect pages during decode
- Discard on writes to protected pages

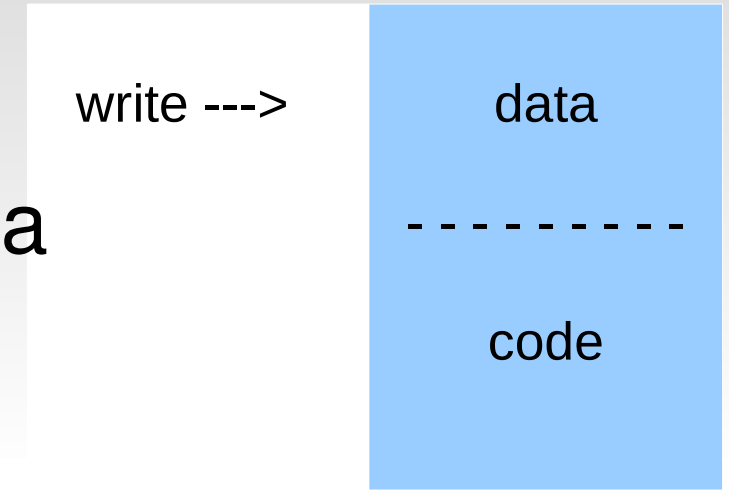
```
decode:
    protect(page, READONLY)
    return translate(fetch(pc))

write fault:
    if (vs->protection[page].readonly)
        ...simulate write fault...
    else
        cache.discard(page, page+PAGESIZE)
        mprotect(page, ~READONLY)
        restart
```

- Works great in many cases: paging, JIT, ...

But... False Sharing

- Application malloc()'s code
- Page has both code and data
- write is slow:
  - Trap
  - Discard valid translation of code
  - Make page writable, perform write
  - Next use of code: make read-only, retranslate
- Sometimes so slow it dominates running time
- If code writes data: complicated infinite loop



Other Cases

- Also slow for:
  - Recompile every 10K instructions
    - e.g., BitBlt()
  - Frequent instruction patching
    - Register numbers, instruction immediates
    - Debugger watchpoints
  - Other fast-changing SMC ”styles”
- I have seen these in commercial workloads...

Optimize For Many Cases

- General strategies
  - Reoptimize: handle the ”new” case fast

    … but no longer handle ”old” case
  - Deoptimize: handle both cases

    … but both cases are slower
  - Keep multiple ”fast” cases + dispatch

    … but ”dispatch” overhead
- Often works

What To ”Trigger” On?

- Instruction events:
  - Write/map event
  - Coherency event (maybe)
  - Execute event
- Simulator events:
  - Lookup
  - Fetch
  - Decode
  - Dispatch
  - Execute

Coherency Events

- x86 (and others): no primitive
  - Need to detect what changed
- Platform ”primitive” for instruction coherency

```
iscp
iflush32 ADDR
coherency(base, length)
```

  - ISCP: ”something changed”
  - Poor match between application and simulator
  - Need to detect what **really** changed

Approach: Try A Strategy Adapt If Too Expensive

- Default: write-protect on translate, fault on write
- Faults are expensive, so...
- After ”too many” faults, try another strategy
  - Asymptotically slower, but avoids faults
- After ”a while” try default strategy again

A Strategy: Self-Checking

```
translate:
    t->original = copy(pc, length)
    t->code.emit(CHECK, t->original, length)
    t->code.emit(TRANSLATE, pc, length)

translation_1234:
    If (miscompare(pc, ORIGINAL, LENGTH))
        return FAIL
    … simulate ....
```

- Polls for coherency
- 2X slower than write-protect → usually avoid
- No READONLY faults
  - Faster on fault-prone code
  - Adaptive: gets used only on fault-prone code

Fast-Changing Code

- Self-checking avoids write faults
- Avoids discard of ”good” translations
- But: need to retranslate all true code changes
  - Frequent changes → high retranslation cost
- Other strategies:
  - Trade off: faster translator, slower code
    - Knob? Multiple translators?
  - Save ”invalid” fast code, see if it reappears
    - Many SMC patterns have just a few values

Adaptive, Take 2

- Same as before, but...
- On self-check failure, save the ”bad” translation
- And: before translating
  - Scan ”bad” translations
  - ”Revalidate” if memory now matches
  - Reuse translations that now work
- ”Revalidating” is cheaper than retranslating

Problem Solved!

- Almost
  - Good: more applications run fast
  - Bad: some are still slow
  - Why: details of SMC/ISC usage
  - E.g., some cases of instruction patching
    - Lots of values for instruction immediates
    - No reuse of earlier translations
- Is it okay if some workloads are slow?
  - Depends on your application

Example: Shade

- Simulates user-space SPARC on SPARC
  - Used for program analysis
  - Performance is ”optional”
  - If it's slow sometimes, that's okay
- Always translates, ~100I/I
- SPARC: iflush *ADDR* signals coherency
- Applications missing iflush:
  - User has to say, via command-line flag
  - Writable memory: ”self-discarding” translations

Example: Crusoe

- Crusoe: commerical x86 CPU: Must be fast!
- Default: protect on translate, discard on write
  - Translation: ~10,000 I/I. Avoid retranslation!
- High fault rate, retranslate (subpage hardware):
  - Write: save translations, make subpage writable
  - Execute: reprotect, revalidate translations
- If *still* high fault rates: retranslate self-checking
- Self-check fails: retranslate: ”fetch immediates”
- Still fails: retranslate: ”call interpreter”
  - What was that about ”avoid retranslation?”

Phew!

- No ”best” strategy
- Depends on the requirements

- A few more notes:

Deoptimize: What Is A ”More General” Translation?

- Fetch instruction immediates
- Translation that calls to the interpreter
- Implements several past instructions
  - Check memory and dispatch accordingly
  - Multiple implementations and dispatch?
  - The translation is dispatching within itself

Oh, And:
It Needs To Work

- Bad: more implementations:

  more bugs

- Worse: more implementations:

  worse coverage of each case

Stability

- Adaptation can ”hunt” endlessly
  - Cost to check and fail
  - Plus cost to adapt
  - Plus cost to execute
- ”Consistent” gets more important than ”fast”
- Sometimes a ”slow” strategy is faster

Conclusion

- SMC/ISC is an important and thorny problem
- Many cases are in a big-enough workload set
- Hard to solve well but:
  - Most cases ”suitably” solvable
  - State clearly what you do and don't do
- Why you want to read the paper:
  - More complicated SMC/ISC cases
  - More strategies
  - More examples of existing systems

Universal Simulator [Gill51]

```
29: A 11  0     # load "load [PC]"
30: A  2  0     # increment PC
31: G  9  0     # goto top
 9: U 11  0     # save "load [PC]" -> 11
10: S 11  0     # clear accumulator
11:[______]     # "load [PC]": *PC -> accumulator
12: U 22  0     # save *PC -> 22

14: S  0        # check for branch...
15: A  4  0     # ...
16: E 19  0     # ... not branch go to 19
  ...           # ... branch: fix "load PC"; goto 9

19: U     0     # clear accumulator
20: S     0     # ...
21: A  1  0     # load vs->accumulator
22:[______]     # execute *PC
23: U  1  0     # save vs->accumulator
24: E 26  0     # branch to 26 if positive
25: A  3  0     # add -1/2 for negative
26: S  1  0     # adjust copy(vs->accumulator)
27: U     0     # save vs->sign
28: S     0     # ...
```