CS 5/6110, Software Correctness Analysis, Spring 2021

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112

Overview of Lecture 19

- Projects
  - Please fix meeting this week (schedule online) to
    - Select/Refine projects
    - Seek resources
      - Projects will be the central aspect of your class
- Today's topic : Cache Coherence

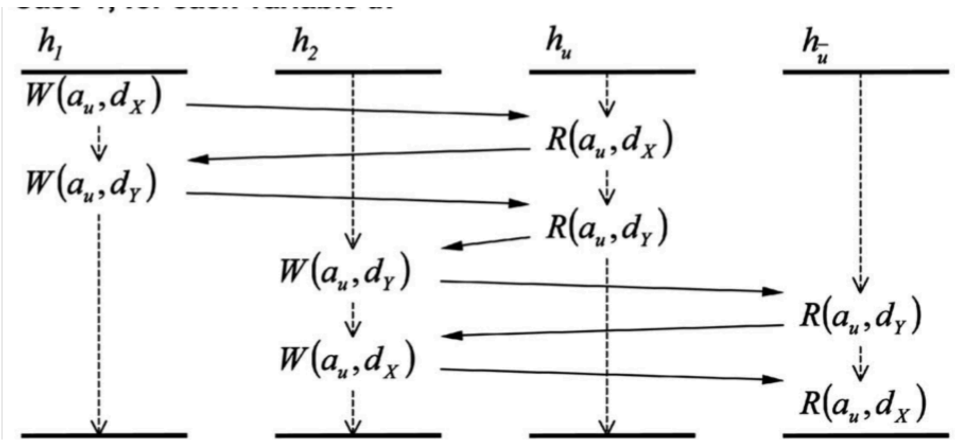
What is cache coherence?

- With shared-memory multiprocessors being the norm, we need a notion of sharing memory “as if it were one common area”
  - What does this mean?
    - Every write by Pi is readable by Pi as well as Pj where i != j
  - This is not precise-enough
    - A program can write more times to a location than once ☺
    - There could be multiple readers of a line
- So,
  - Formally define when write-events are observable by read-events that match

What is cache coherence?

- To define memory views as shared by multiple processors, we need to define a formal shared memory consistency model
- Coherence is one of the basic models
  - Each location has a latest data that every reader agrees on
  - Also known as “per-location sequential consistency”
- There are some interesting complexity results that help us understand coherence
  - Given a “finished execution trace”, checking coherence is NP-Complete
    - Very insightful proof by Jason Cantin et al
    - https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1435343

How a programmer sees coherence (Cantin paper)



Given four processes (or hardware threads) and reads/writes
Per location ("a_u" in this case) explain read-value outcomes.
Here we explain as if this interleaving took place.
The inability find such an explanation means the system is incoherent
Cool piece of encoding by Cantin: Given a 3SAT formula, he generates a shared memory execution such that the formula is SAT iff the execution is coherent!

If you are looking forward to challenging yourself in your growth as a professional CS researcher, you should not pass up such opportunities (follow such proofs). I can help if you are interested!

When you finish this, you'll feel a faint glow around your head ;-)
ASIDE: There is a panel on "how to become a faculty" tomorrow. I'd say that solving these kinds of puzzles can help you!
i.e. before you get out of the SoC, try to get the union of knowledge of the SoC faculty
* Getting the intersection is a bad idea → you'll get close to the null set ☺

Implementing coherence

- Snoopy-bus protocols (still present at smaller scales)
- Directory-based protocols (more scalable)
- See https://www.morganclaypool.com/doi/pdf/10.2200/S00962ED2V01Y201910CAC049 for Thu’s colloq speaker’s book

Coherence Verification

- Given the complexity of coherence protocols, formal methods (model-checking mainly) is essential
- Let us take a look at an academic protocol called The German Protocol
- How does coherence verification scale?
  - Not well – today, large protocols take days to cover for one bit of data and 3 processors
  - Solutions
    - Derive cutoff bounds – Emerson and Kahlon
      - The bounds are large (7-8) and automatically computing bounds is not practical for large protocols
    - Do a parametric verification
      - Prove coherence for all "N" – N is the number of cores/threads
      - This involves modeling 2-3 nodes explicitly and involves a manual abstraction/refinement loop – called CEGAR
        - Counter-Example Guided Abstraction Refinement
      - Involves designer input of non-interference lemmas
    - Do formal synthesis
      - Dr. Nagarajan will be presenting this on Thu

Basics about Transition Systems

- Before we study the German protocol and how the parametric verification method I’m going to present works, let us discuss basic concepts about formal transition systems

Almost all specs for safety property checking look like this

- Based on Guarded Commands

  Rule1:   g1 ==> a1

  Rule2:   g2 ==> a2

  …

  RuleN: gN ==> aN

  Invariant P

- Supported by tools such as Murphi (Stanford, Dill's group)
- Presents the behavior declaratively
  - Good for specifying "message packet" driven behaviors
  - Sequentially dependent actions can be strung using guards
- "Rule Sets" can specify behaviors across axes of symmetry
  - Processors, memory locations, etc.
- Simple and Universally Understood Semantics

Let us understand how we may safely transform such rule-based specifications (this is what we have to do in the parametric verification approach to be presented). The method is called the CMP method named after its inventors at Intel.

The name of the game – “invariants” !!

- Much like loop invariants, except
  - These are invariants pertaining to formal state-transition systems
- We are really talking about inductive invariants
  - Invariants P such that
    - They are true in the initial state
    - When held at a state, every transition rule preserves it
- This is a hugely important topic (inductive invariant discovery)
  - Then you can do program-proofs by walking each path or transition rule!

Invariants, Inductive Invariants

- Consider the transition system
  - Init : X == 1 (X is an int var … i.e. +, 0, -)
  - Tr1 : true ➔ x += 2
  - Tr2 : true => x += 3
  - An invariant is x != 0
  - But is it inductive?
    - (X != 0) ==?==Tr1➔ (X!=0) ?
    - (X != 0) ==?==Tr2➔ (X!=0)?

Invariants, Inductive Invariants

- Consider the transition system
  - Init : X == 1 (X is an int var … i.e. +, 0, -)
  - Tr1 : true ➔ x += 2
  - Tr2 : true => x += 3
  - An invariant is x != 0
  - But is it inductive?
    - (X != 0) ==?==Tr1➔ (X!=0) ?
    - (X != 0) ==?==Tr2➔ (X!=0)?
  - No, for X == -2, it is not true!
    - So what is one inductive invariant for this transition system?

Invariants, Inductive Invariants

- Consider the transition system
  - Init : X == 1 (X is an int var … i.e. +, 0, -)
  - Tr1 : true ➔ x += 2
  - Tr2 : true => x += 3
  - An invariant is x != 0
  - But is it inductive?
    - (X != 0) ==?==Tr1➔ (X!=0) ?
    - (X != 0) ==?==Tr2➔ (X!=0)?
  - No, for X == -2, it is not true!
    - So what is one inductive invariant for this transition system?
      - How about X >= 0 ?
      - How about X >= -1 ?
      - How about X >= 1 ?
      - …
    - What is the strongest inductive invariant of the above three?
    - What is the strongest inductive invariant (period) ?
      - Answer : = reachable state space!

New cool find today

- New cool find: The I4 tool, and this hotos paper !
  - https://web.eecs.umich.edu/~manosk/assets/papers/i4-hotos19.pdf
- https://web.eecs.umich.edu/~manosk/projects.html
- https://web.eecs.umich.edu/~manosk/assets/papers/i4-hotos19-slides.pdf

- I AM SUPER-EXCITED – A PROJECT to read and reconstruct
- https://researchr.org/publication/fmcad-2021 (automated paxos proof ?!)

Let us understand how we may safely transform such rule-based specifications. The first few will be warmups. Then the real thing!

- Observation: Weakening a guard is sound
- Suppose we add a disjunct as below (Cond1) and still manage to show that P is an invariant, then without adding Cond1, the result must still hold

  Rule1: g1 \/ Cond1 ==> a1

  Rule2: g2 ==> a2

  Invariant P

- Reason: Rule1 fires more often with Cond1 added!
- May get false alarms (**P** may fail if Rule1 fires spuriously)
- For many "weak properties" **P**, we can "get away" by guard weakening
  - This is a standard abstraction, first proposed by Kurshan (E.g. removing a module that is driving this module, letting inputs "dangle")
- BUT in the CMP method, we won't do this – rather we will do guard strengthening!
- Except it is useful to know this disjunction property in thinking about certain steps of CMP

But… Guard Strengthening is, by itself, Unsound

- Strengthening a guard is not sound

  Rule1: g1 /\ Cond1 ==> a1

  Rule2: g2 ==> a2

  Invariant P

- Reason: Rule1 fires only when **g1** /\ **Cond1**
- So, less behaviors examined in checking **P**
  - **Thus, verifying _with_ Cond1 means nothing for verification without Cond1**
- **But hang on, there is more in the CMP method ☺**

Guard Strengthening can be made sound, if the conjunct is implied by the guard

- This is sound
  - **Rule1: g1 /\ Cond1 ==> a1**
  - **Rule2: g2 ==> a2**
  - **Invariant P /\ (g1 ==> Cond1)**
- Reason: Rule1 fires only when **g1** /\ Cond1
- BUT, **Cond1** is always implied by **g1; we are showing g1➔Cond1 is an invariant also** , so no real loss of states over which Rule1 fires…
  - Call this “Guard Strengthening Supported by Lemma”
- Except, you are showing the invariant in the same modified system!
  - This is fine:
    - Initial state satisfies P, and also (g1 ➔ Cond1). Thus, whenever g1 is true in the initial state, Cond1 is an implied fact. So g1 /\ Cond1 is like g1 by itself.
    - Thus “a1” can be conducted in the initial state , to obtain the next set of states. (No change wrt g2 and a2, so they are like before.)
  - In general
    - At state t (by induction over time), we have P true and (g1 ➔ Cond1) true.
    - Thus , g1, when true, implies Cond1 at time t. Thus g1 /\ Cond1 is like g1 (same truth status) at time t
      - Thus we can obtain the state at t+1 safely via “a1”

Summary of Transformations so far (checkmark shows what's safe)

rule g1 ==> a1;

rule g2 ==> a2;

invariant P;

---

rule g1 $\wedge$ cond1 ==> a1;

rule g2 ==> a2;

invariant P;

X

---

rule g1 $\vee$ cond1 ==> a1;

rule g2 ==> a2;

invariant P;

✓

---

rule g1 $\wedge$ cond1 ==> a1;

rule g2 ==> a2;

invariant P $\wedge$ (g1 => cond1);

✓

The CMP Approach

- Weaken to the Extreme
- Then Strengthen Back Just Enough (to pass all properties)

Weaken to the Extreme sounds crazy at first!

Rule1: g1 \/ True ==> a1

Rule2: g2 ==> a2

Invariant P

The transition system above

can be transformed to the one below without any issues

(except the proof of P being an invariant might not go through) – but that will be fixed momentarily

Rule1: True ==> a1

Rule2: g2 ==> a2

Invariant P

Strengthen Back Some

Rule1: True /\ C1 ==> a1
Rule2: g2 ==> a2
Invariant P /\ (g1 => C1)

"Not Enough!" may be
the outcome of strengthening.
That is, while we added C1 back,
it may not be strong-enough.

How to pick C1 will be discussed soon.

Strengthen Back More...

Rule1: True /\ C1 ==> a1
Rule2: g2 ==> a2
Invariant P /\ g1 => C1

"Not Enough!"

Rule1: True /\ C1 /\ C2 ==> a1
Rule2: g2 ==> a2
Invariant P /\ (g1 => C1) /\ (g1 => C2)

"OK, just right!"

A Variation of Guard Strengthening Supported by Lemma: Doing it in a meta-circular manner (i.e., the temporal induction I alluded to earlier…)

rule g1 ==> a1;
rule g2 ==> a2;
invariant P;

→

rule g1 /\ cond1 ==> a1;
rule g2 ==> a2;
invariant P /\ (g2 => cond2);
rule g1 ==> a1;
rule g2 /\ cond2 ==> a2;
invariant P /\ (g1 => cond1);

✓

This is the approach in our work

Now the secret: in the CMP method, the designer decouples all nodes beyond k (typically 2) explicitly modeled nodes. Then, brings back the N-k nodes, but in ‘spirit’ – i.e., in terms of the non-interference conditions they must obey (note that “N” is a free parameter).

The Ci are thus the non-interference lemmas. Each is discovered upon seeing a counterexample, and then added back into the system! If the coherence invariant is proved (usually this happens), then you ended up having used a model-checker to prove a parametric theorem – which is a big deal!

```
rule g1 ==> a1;

rule g2 ==> a2;

invariant P;
```

→

```
rule g1 /\ cond1 ==> a1;

rule g2 ==> a2;

invariant P /\ (g2 => cond2);
```

```
rule g1 ==> a1;

rule g2 /\ cond2 ==> a2;

invariant P /\ (g1 => cond1);
```

✓

This is the approach in our work

This method has been perfected at Intel and in production use!

- See https://www.cs.utexas.edu/~hunt/FMCAD/FMCAD09/slides/talupur.pdf
- https://dl.acm.org/doi/pdf/10.5555/1517424.1517434
- Designers write "protocol flow" diagrams as part of standard documentation
  - These are mined to obtain the non-interference lemmas
- See http://formalverification.cs.utah.edu/presentations/fmcad04_tutorial2/chou/ctchou-tutorial.pdf for details of how this is done
- NOTE the table-style specification recommended in the above tutorial at fmcad04 !!

We will now present highlights of the German Protocol to tell you how coherence protocols look like (but this is like a “hello world” of cache protocols)

- See german.m , german.pdf , and abs-german.pdf in the class directory
- We will note down some highlights in the coming slides

CMP highlights from paper of C-T Chou

```
abs-german.m          Mon Nov  1 20:00:47 2004          1

const   ---- Configuration parameters ----

  NODE_NUM : 2;
  DATA_NUM : 2;

type    ---- Type declarations ----

  NODE : scalarset(NODE_NUM);            -- NODE now consists of
                                         -- only concrete nodes
  DATA : scalarset(DATA_NUM);

  ABS_NODE : union {NODE, enum{Other}}; -- ABS_NODE consists of both
                                         -- concrete and abstract nodes
```

CMP highlights from paper of C-T Chou

We now do the model checking, which produces the following counterexample to CtrlProp: node n_1 sends a ReqE to home; home receives the ReqE and sends a GntE to node n_1; node n_1 receives the GntE and changes its cache state to E; node n_2 sends a ReqS to home; home receives the ReqS and is about to send an Inv to node n_1; but suddenly home receives a bogus InvAck from Other (via ABS_RecvInvAck), which causes home to reset ExGntd and send a GntS to node n_2; node n_2 receives the GntS and changes its cache state to S, which violates CtrlProp because node n_1 is still in E. The bogus InvAck from Other is clearly where things start to go wrong: if there is a node in E, home should not receive InvAck from any other node. We can capture this desired property as a ***noninterference lemma***:

```
invariant "Lemma_1"
  forall i : NODE do
    Chan3[i].Cmd = InvAck & CurCmd != Empty & ExGntd = true ->
    forall j : NODE do
      j != i -> Cache[j].State != E & Chan2[j].Cmd != GntE
  end end;
```

CMP highlights from paper of C-T Chou

which says that if home is ready to receive an `InvAck` from node `i` (note that the antecedent is simply the precondition of `RecvInvAck` plus the condition `ExGntd = true`, which is the only case when the `InvAck` is to have any effect in `ABS_RecvInvAck`), then every other node `j` must not have cache state `E` or a `GntE` in transit to it. (We are looking ahead a bit here: if the part about `GntE` is omitted from `Lemma_1`, the next counterexample will compel us to add it.) If `Lemma_1` is indeed true in GERMAN, then we will be justified to ***refine*** the offending abstract ruleset `ABS_RecvInvAck` as follows:

CMP highlights from paper of C-T Chou

```
rule "ABS_RecvInvAck"
  CurCmd != Empty & ExGntd = true
==>
  ExGntd := false; undefine MemData;
end;
```

The top-left rule is changed to the bottom-right rule in the CMP method, by introducing non-interference lemmas to strengthen the guards. These lemmas are proven in the same model being refined.

```
rule "ABS_RecvInvAck"
  CurCmd != Empty & ExGntd = true &
  forall j : NODE do
    Cache[j].State != E & Chan2[j].Cmd != GntE
  end
==> ... end;
```

German protocol

```
const -- configuration parameters --
NODE_NUM : 6;
DATA_NUM : 2;

type -- type decl --

NODE : scalarset(NODE_NUM);
DATA : scalarset(DATA_NUM);

CACHE_STATE : enum {I, S, E};
CACHE : record State : CACHE_STATE; Data : DATA; end;

MSG_CMD : enum {Empty, ReqS, ReqE, Inv, InvAck, GntS, GntE};
MSG : record Cmd: MSG_CMD; Data : DATA; end;
```

German protocol

```
var -- state variables --

Cache : array [NODE] of CACHE;     -- Caches

Chan1 : array [NODE] of MSG;       -- Channels for Req*
Chan2 : array [NODE] of MSG;       -- Channels for Gnt* and Inv
Chan3 : array [NODE] of MSG;       -- Channels for InvAck

InvSet : array [NODE] of boolean; -- Nodes to be invalidated
ShrSet : array [NODE] of boolean; -- Nodes having S or E copies
ExGntd : boolean;                  -- E copy has been granted
CurCmd : MSG_CMD;                  -- Current request command
CurPtr : NODE;                     -- Current request node
MemData : DATA;                    -- Memory data
AuxData : DATA;                    -- Latest value of cache line
```

German protocol

```
-- Initial States --

ruleset d : DATA do startstate "init"
--
-- All nodes: init all cmd channels to be empty, Cache States I,
-- the set of nodes to be invalidated is empty
-- and nodes having S or E copies empty
--
  for i : NODE do
    Chan1[i].Cmd := Empty;
    Chan2[i].Cmd := Empty;
    Chan3[i].Cmd := Empty;
    Cache[i].State := I;
    InvSet[i] := false;
    ShrSet[i] := false;
  end;
  ExGntd := false;
  CurCmd := Empty;
  MemData := d;
  AuxData := d;
end end;
```

German protocol

```
-- State Transitions --
---------------------------------------------
ruleset i : NODE do
-- Any node with cmd req channel empty and cache I can request ReqS
  rule "SendReqS"
    Chan1[i].Cmd = Empty &
    Cache[i].State = I
    ==>
    Chan1[i].Cmd := ReqS;  -- raises "ReqS" semaphore
  end
end;
```

German protocol

```
------------------------
ruleset i : NODE do
-- Any node with cmd req channel empty and cache I/S can request ReqE
  rule "SendReqE"
    Chan1[i].Cmd = Empty &
   (Cache[i].State = I |
    Cache[i].State = S)
    ==>
    Chan1[i].Cmd := ReqE;  -- raises "ReqE" semaphore
  end
end;
```

German protocol

```
----------------------------------------
ruleset i : NODE do
-- For any node that is waiting with ReqS requested, with CurCmd Empty
-- we set CurCmd to ReqS on behalf of node i (setting CurPtr to point to it).
-- Then void Chan1 empty.
-- Now Set the nodes to be invalidated to the nodes having S or E copies.
  rule "RecvReqS" -- prep action of dir ctrlr
    CurCmd = Empty &
    Chan1[i].Cmd = ReqS
    ==>
    CurCmd := ReqS;
    CurPtr := i; -- who sent me ReqS
    Chan1[i].Cmd := Empty;         -- drain its cmd
    for j : NODE do InvSet[j] := ShrSet[j] end; -- inv = nodes with S/E
  end
end;
```

German protocol

```
-----------------------
ruleset i : NODE do
-- For any node that is waiting with ReqE requested, with CurCmd Empty
-- we set CurCmd to ReqE on behalf of node i (setting CurPtr to point to it).
-- Then void Chan1 empty.
-- Now Set the nodes to be invalidated to the nodes having S or E copies.
  rule "RecvReqE"
    CurCmd = Empty &
    Chan1[i].Cmd = ReqE
    ==>
    CurCmd := ReqE;
    CurPtr := i; -- who sent me ReqE
    Chan1[i].Cmd := Empty;       -- drain its cmd
    for j : NODE do InvSet[j] := ShrSet[j] end; -- inv = nodes with S/E
  end
end;
```

German protocol

```
----------------------------------------
ruleset i : NODE do
-- For every node with Chan2 Cmd empty and InvSet true (node to be invalidated)
-- and if CurCmd is ReqE or (ReqS with ExGnt true), then
-- void Chan2 Cmd to Inv, and remove node i from InvSet (invalidation already set out)
  rule "SendInv"
    Chan2[i].Cmd = Empty &
    InvSet[i] = true &  -- Gnt* and Inv channel
    ( CurCmd = ReqE |                      -- DC: curcmd = E
      CurCmd = ReqS & ExGntd = true ) -- DC: curcmd = S & ExGntd
  ==>
    Chan2[i].Cmd := Inv; -- fill Chan2 with Inv
    InvSet[i] := false;
  end
end;
```

German protocol

```
----------------------
--
-- When a node gets invalidated, it acks, and when it was E
-- then the node (i) coughs up its cache data into Chan3
-- Then cache state is I and undefine Cache Data
--
ruleset i : NODE do
  rule "SendInvAck"
    Chan2[i].Cmd = Inv &
    Chan3[i].Cmd = Empty
    ==>
    Chan2[i].Cmd := Empty;
    Chan3[i].Cmd := InvAck;
    if (Cache[i].State = E) then Chan3[i].Data := Cache[i].Data end;
    Cache[i].State := I; undefine Cache[i].Data;
  end
end;
```

German protocol

```
-----------------------
ruleset i : NODE do
  rule "RecvInvAck"
    Chan3[i].Cmd = InvAck &
    CurCmd != Empty
    ==>
    Chan3[i].Cmd := Empty;
    ShrSet[i] := false;
    if (ExGntd = true) then ExGntd := false;
    MemData := Chan3[i].Data;
    undefine Chan3[i].Data end;
  end
end;
```

German protocol

```
-----------------------------------------
ruleset i : NODE do
  rule "SendGntS"
    CurCmd = ReqS &
    CurPtr = i &
    Chan2[i].Cmd = Empty &
    ExGntd = false
    ==>
    Chan2[i].Cmd := GntS;
    Chan2[i].Data := MemData;
    ShrSet[i] := true;
    CurCmd := Empty;
    undefine CurPtr;
  end
end;
```

German protocol

```
----------------------
ruleset i : NODE do
  rule "SendGntE"
    CurCmd = ReqE &
    CurPtr = i &
    Chan2[i].Cmd = Empty &
    ExGntd = false &
    forall j : NODE do ShrSet[j] = false end -- nodes having S or E status
    ==>
    Chan2[i].Cmd := GntE;
    Chan2[i].Data := MemData;
    ShrSet[i] := true;
    ExGntd := true;
    CurCmd := Empty;
    undefine CurPtr;
  end
end;
```

German protocol

```
------------------------
ruleset i : NODE do
  rule "RecvGntS"
    Chan2[i].Cmd = GntS
    ==>
    Cache[i].State := S;
    Cache[i].Data := Chan2[i].Data;
    Chan2[i].Cmd := Empty;
    undefine Chan2[i].Data;
  end
end;
```

German protocol

```
----------------------
ruleset i : NODE do
  rule "RecvGntE"
    Chan2[i].Cmd = GntE
    ==>
    Cache[i].State := E;
    Cache[i].Data := Chan2[i].Data;
    Chan2[i].Cmd := Empty;
    undefine Chan2[i].Data;
  end
end;
```

German protocol

```
----------------------
ruleset i : NODE;            -- for every node i
        d : DATA             -- for every data d
  do
    rule "Store"
      Cache[i].State = E     -- if node is in E
      ==>
      Cache[i].Data := d;    -- store d into Cache[i].Data
      AuxData := d;          -- Also update latest cache line value
    end                      -- The node in E can get any "D" value
end;

---------------------------------------------
```

Invariants of the German protocol

```
--------------------------------------------


---- Invariant properties ----

invariant "CtrlProp"
forall i : NODE do
  forall j : NODE do
   i!=j ->
    (Cache[i].State = E -> Cache[j].State = I) &
    (Cache[i].State = S -> Cache[j].State = I |
                           Cache[j].State = S)
  end
end;

invariant "DataProp"
( ExGntd = false -> MemData = AuxData ) &
 forall i : NODE
 do Cache[i].State != I ->
    Cache[i].Data = AuxData
 end;


--------------------------------------------
```