CAV: An Industrial Perspective

Robert Kurshan
July 2015

Design Costs

- synthesis/layout < 50% total cost
  - more or less linear in chip size
- debug/verification = 50% - 80% of total cost
  - embedded software
  - n parallel components of size m leads to m^n system states
  - so functional verification grows exponentially with design size
- widely held that the cost of fixing a bug grows exponentially with the development stage at which it is detected/fixed
  - on account of increasing interactions with other components that also must reflect changes from fixes
- holding down development costs can lead to less test coverage and lower design reliability
  - but cost of product failures can also be high
  - ready for a $500M recall? (Intel FDIV bug)
  - How about $2B? (A number of automotive recalls are due to s/w bugs)

Formal Commercial Usage Today

- Generic functionality
  - EC
  - Arbitration
  - Resource allocation (request/grant properties)
  - Flow control (underflow/overflow)
  - State unreachability
  - X-propagation
  - Reachability of cover points
  - Race conditions
  - Combinational cycles
  - Local message delivery
  - Local serialization
  - Connectivity
  - Security (finding known flaws in new code)
- S/W Drivers (2011 CAV Award)
- Protocols
  - ARM, AMBA, AXI, ACE, PCI,..

Usage Today (cont.)

- FSMs
- Low Power
- CDC
- Memory systems
- Complete block-level verification
- Automatic property generation (eg, register map validation)
- Integration with simulation
- Sudoku, Rubik's cube, chess end-games :-)

NB: Very targeted usages, *not* the earlier ideal of checking vs a system spec.

- Used today by virtually every major chip manufacturer
- Supported by the major EDA vendors
  - Synopsys, Cadence, Mentor, OneSpin, TVS, ++
- Fastest growing EDA product (for last 5-10 yrs).

How We Got There

- Once upon a time, there was no “Formal” in industry
- Had to overcome inherent barriers:
  - any prototype must compete against honed engineering (here, simulation test)
  - vicious circle of technology transfer: transfer requires proof, which requires transfer

A brief history

- axiomatic reasoning (Euclid (300 BC); Russell-Whitehead (1913), ..)
- program correctness: von Neumann-Goldstine (1947)
- linear-time model checking: Church (1957) (!)
- Turing Machine (1936)
- automata theory
  - Strings: Rabin-Scott (1959)
  - Sequences: Buchi: (1962)
- correct-by-construction programs: Dijkstra (1960s)
- formal definition of 'program correctness': Naur (1966); Floyd (1967); Hoare (1969)
- automated proof-checking applied to programs: Boyer-Moore *et al*; Milner (1970s)

Red = Turing award winner

History (cont)

- reasoning about concurrent processes: Lamport (1970s)
- checking LTL properties of non-terminating concurrent programs: Pnueli (1977)
- analysis via state space exploration: West (1978)
- (branching-time) *model checking*: Clarke-Emerson (1980); Queille-Sifakis (1982)
- automata-theoretic verification: Kurshan; Vardi-Wolper (1986)
- automated abstraction: Kurshan *et al* (1990); Clarke *et al* (2003)
- symbolic model checking: McMillan (1993)
- SAT-based model checking: Clarke *et al* (1999)
- interpolation: McMillan (2003)
- cooperating engines: Brayton, Mishchenko (2010)
- inductive strengthening in SAT: Bradley (2011)

Interplay Between Theory and Application

A test of a field's maturity and robustness is the extent to which its applications "pay back" dividends in the form of new theoretical problems, and cross-fertilization with other fields.

- SAT: 100 variables -> 10^5 variables (2009 CAV Award)
  - stimulated by the demands of hardware design
- Interpolation likewise (2010 CAV Award)
- Timed automata (2008 CAV Award)
- Alternating tree automata and parity games
  - stemming from practical questions about equivalence and synthesis
- Push-down automata
- Artificial intelligence
  - model checking contributing to planning and benefitting from machine learning

Technology Transfer: Initial Challenges

A new technology, by definition, is an alien in the environment of the problems it seeks to solve.

- Toy problems needed to get going
- But then, real problems are needed to demonstrate value
  - Access to real problems requires professional implementations
    - Significant investment by both proponent and client
    - Need to penetrate complex client environments
  - Need for implementation to scale: capacity and speed
- Client resistance to change
  - High cost of any significant methodology change
  - Uncertainty about real value of the new technology
  - Uncertainty about particular proponent
    - Is this the best proponent for the client to work with?
  - Time-line: will it converge on my watch?
    - Assuming it's successful, what's the time line to go from prototype to in-flow?
      - Retrofitting flows, retraining, hiring enough experts, …

Initial Challenges (cont.)

- Transfer requires sufficient pain with current practice to warrant costs/uncertainties
- Moving from a simulation to a model checking mindset
  - Requires re-education
  - Is writing properties really easier than writing a test bench?
- Establish the right scope for a prototype
  - System level verification gives the most benefit, but is a big methodological step
  - Block level verification is simpler, but demonstrates more modest benefits
  - Where's the sweet spot???

Settling On a Scope for a Model Checking Prototype

- Walk before you run
- Settle on modest small steps
- Less dramatic can be more convincing
- Apply first to a few small blocks
  - Amenable to automated decision procedures
  - Capacity, speed less of an issue
- But in stark contrast to simulation test of entire design
- Properties needed for blocks
- Constraints needed to model environment

Challenges of Block Level Verification

- Writing Properties/Constraints fell to Verification Team
  - But they don't know block level design functionality
  - Moreover, need to know functionality of adjacent blocks
  - Designers too busy to provide info (or have moved on)
    - Absent designers also has consequences for simulation debug, BTW
  - Need data base of proof obligations (complex infrastructure)
- Optimally done at design time
  - Puts block level debug in the hands of the expert (the architect)
  - Gives designer a powerful debug tool: model checker
  - Verification Team could be retrained to work with designer here
  - Known to greatly speed up flow, when debug is taken into account (Recall: debug now accounts for as much as 80% of design cost.)
  - But: good luck. (It's a disruptive change.)

First Successes

- Led by sense of urgency due to pain of diverging debug cycles
- Required: enlightened management + resources
- But success led to need to compare tools
  - Justify investment by using best available technology
- Demand for common interface for model checker
  - Ie, standardized specification language
- Enter Accellera
  - Idea from model checking
  - But driven by same need with simulation

Challenges Overcome

To successfully transfer a disruptive technology:

1. Break the vicious circle of funding (transfer requires funding that requires justification that requires transfer)
2. Must interface the new technology to the client environment
3. Limit and then integrate methodology changes into client practice
4. Demonstrate the cost-effectiveness of the new technology in the client's environment
5. Enable competitive evaluations.

A Framework for Technology Transfer

- Technology transfer was achieved through a succession of small, incremental steps
  - Small steps limited disruption, cost
  - But must be significant enough to show (some) benefit
- The cut point is near-term cost-effectiveness
  - the small step needs only to provide a short-term benefit greater that its adoption cost
  - Longer-term benefits are too hard to predict and thus are generally heavily discounted
- Still, hard to get right
  - Which small steps?
  - Do they eventually lead to the goal?
- The real guide was trial and error
- The hidden angel: sharply increasing pain of simulation alone

Cycles of Employment and Academic Support

- 1990-1995 – “formal” was *hot*: “sexy” new area, attracted lots of students, funding (SRC, NSF, …)
- 1995-2005 – where are the jobs?? The pull from industry had not yet matched the push from academia. Students went to other areas; funding dried up.
- 2005-2015 – The pull from industry (finally!) surged. Where are the students?? Very hard to fulfill demand for good post-docs with “formal” background. EDA cannot keep up with the demand. And it’s getting worse: recognized dire need to verify embedded s/w, general s/w, and yes: system level correctness.

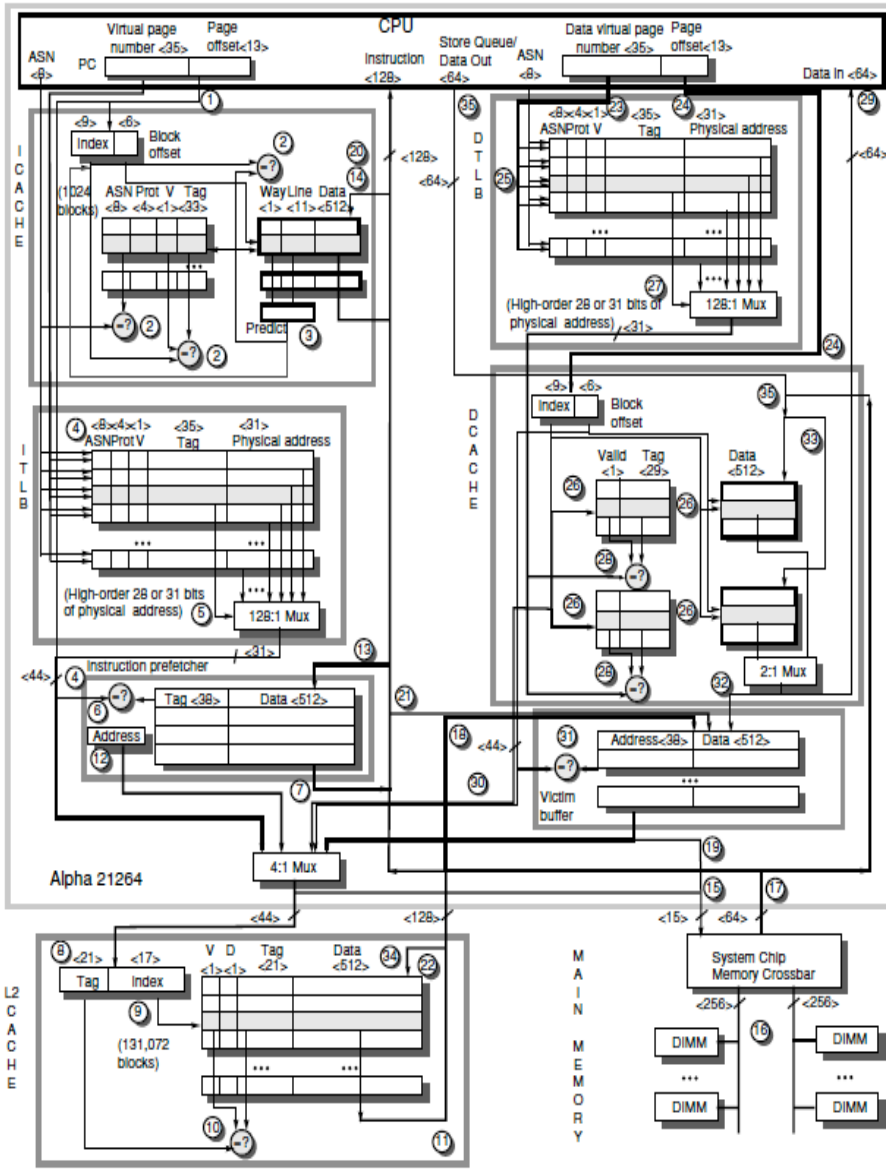
Left On The Table

Readily providable through current technology

- Full integration with simulation
- Regression check-sum data base
  - Your EDA vendor doesn't want you to know this!
- Full AG management
- cache system verification
  - Verify memory consistency, not cache protocol!

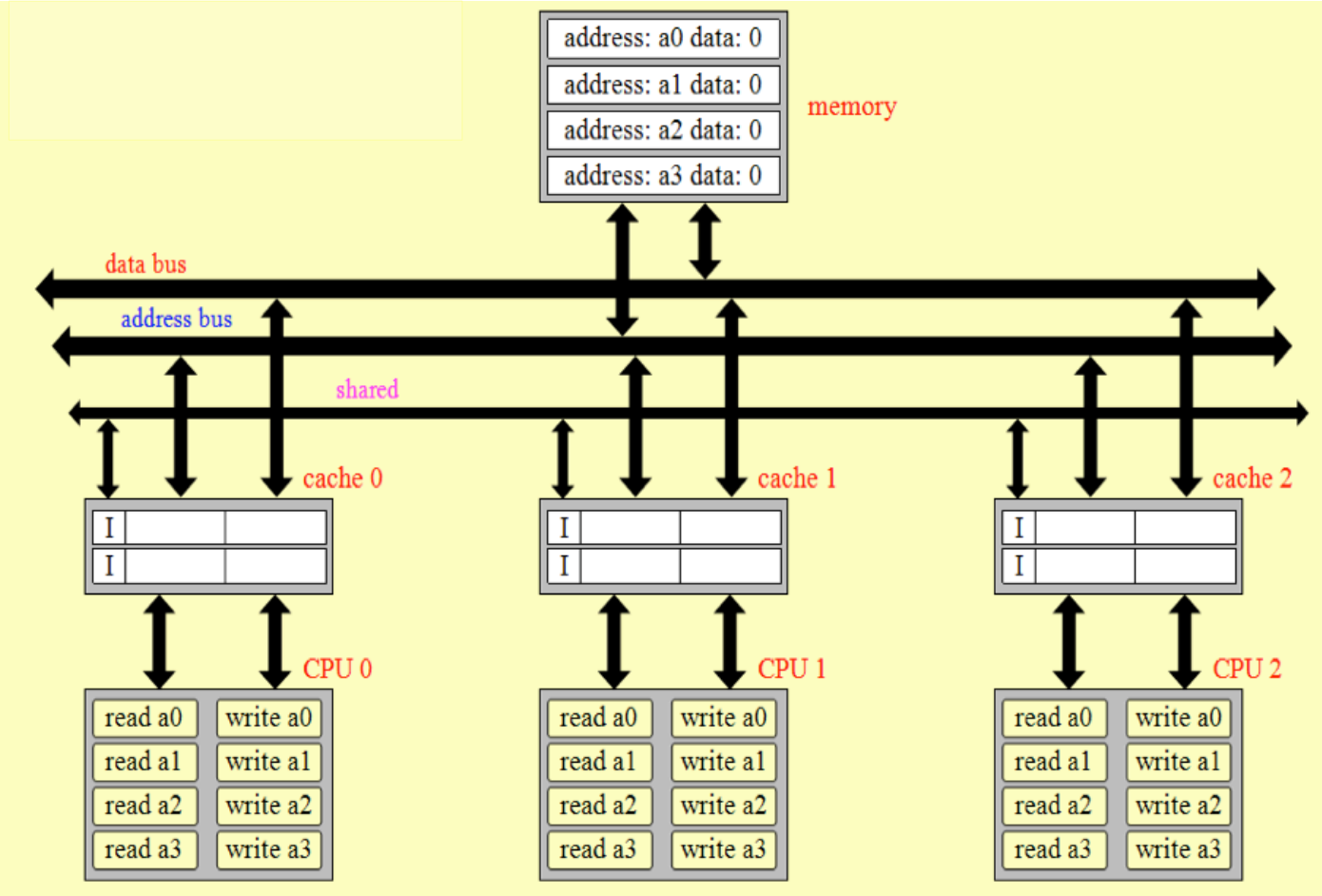
Verifying Memory Consistency

Alpha 21264 Memory Hierarchy



Hennessy, Patterson “Computer Architecture”

Distributed Caches



http://www.scss.tcd.ie/Jeremy.Jones/vivio/caches/MESI.htm

The Goal: “Consistency of Memory”

All processes should “see the same values of data”.

(whatever that means!)

Memory Consistency Models: serializability

Serializable = Linerarizabile (= Atomizable = Sequentially Consistent [H-P])

- reads and writes can be totally ordered so that any read(A) returns the last value of write(D, A)

[NB: little uniformity of terminology!]

nice for programmers and compilers, but implementation is too slow, requiring higher bandwidth on databus and higher latency than other workable schemes that are less restrictive

notation:

P, Q, R: asynchronous processes

A: memory address

D: data word

[H-P] = Hennessy, Patterson "Computer Architecture"

Memory Consistency Models: weak consistency

weak (sequential) consistency (aka "sequential consistency"; implements "program order preservation")

[WC] P.write(D,A), not(Q.write(.,A))*, P.read(A) = X => X = D

captures the sequential sanity expected from a sequential program P

note: if we add the [H-P] "data migration" property:

[DM] P.write(D,A), not(Q.write(.,A))*, R.read(A) = X => X = D
provided time(not(Q.write(.,A))*) > Δ

then [WC] + [DM] => serializablity, provided successive reads and writes to the same address are at least time Δ apart

(NB: [H-P] doesnt get this - and a number of other points - quite right.)

Memory Consistency: weak consistency (cont.)

weak consistency allows for counter-intuitive behaviors:

P: X=0	Q: Y=0
...	...
X=1	Y=1
if(Y==0)..	if(X==0)..

In a serializable memory model, having both if()=true is not possible.

However, under WC both may be true at the same time: the write-invalidate(X) may not have reached Q by the time it tests if(X==0).

CRAZY!

Memory Consistency Models: Weak Data Migration

To counteract the chaos of weak consistency, designers require certain special additional inter-process requirements usually cast in terms of "synchronization" variables or later, memory “barriers”. For simplicity, will consider synchronization variables.

For reasons that will become apparent, for synchronization variables, I choose to cast these requirements in terms of a **Weak Data Migration** property:

[WDM] P.write(D,A), not(Q.write(.,A))*, R.read(A) = X => X = D

for a (small) set of "synchronization variables" A

In practice, synchronization variables are written only by special synchronization processes ("synchronizers") at special times (when needed for synchronization) in a manner that avoids a race between P.write(D,A) and R.read(A), through locks, fifos or other means that prevent a read from returning with the "wrong" synchronization data.

Memory Consistency Models: Memory Coherence

To avoid the terminology chaos, for the case of synchronization variables, I'll invent a new term:

[MC] **Memory Coherence** = Weak Consistency + Weak Data Migration

Memory barriers move some of the synchronization programming into the hardware. They simplify programming and complicate verification; a small set of "data migration" properties can be expressed for memory barriers as well, that together with [WC] give an associated definition of memory coherence [MC']. Memory barriers could be programmed with synchronization variables, and thus

[MC] => [MC'].

Memory Consistency Models:
"Memory Coherence = Weak Consistency" !

[MC] has been widely used (although generally not recognized in just this way).

It allows for just enough quantifiable Data Migration to support a microprocessor that's reasonably programmable and yet efficient in terms of performance and bandwidth consumed on the databus.

Something else very nice about [MC] (but not generally recognized) is that verifying [MC] generally can be reduced to verifying [WC] for a modified process definition: generally, on account of the way synchronizers work, for any process P and its synchronizer processes S, the pair (P, S) behaves like a synchronous program. Therefore we may replace every process P with the pair (P, S). Since only S modifies synchronization variables,

Memory Coherence = Weak Consistency

over the modified process space.

Implementing Memory Consistency

two layers:

memory consistency model ("*the what*")

cache coherence protocol ("*the how*")

- each may be checked in an abstract model or RTL
- each may be checked by guided-random simulation based on families of scenarios
- each may be checked by model checking based on symmetry reduction

Memory consistency model: I focused here on Memory Coherence (although, there are many variations and relaxations: [MC'], "release-", "weak-", "causal-", "FIFO-" consistency)

The MESI ("Illinois") Cache Coherence Protocol

Each cache line is in one of four states:

Modified - cache line is dirty

must be written back to main memory (happens via a "writeback") before another proc can read it; then

-> *Exclusive*

Exclusive - cache line is clean (ie, = value in main memory)

-> *Shared* upon read-request

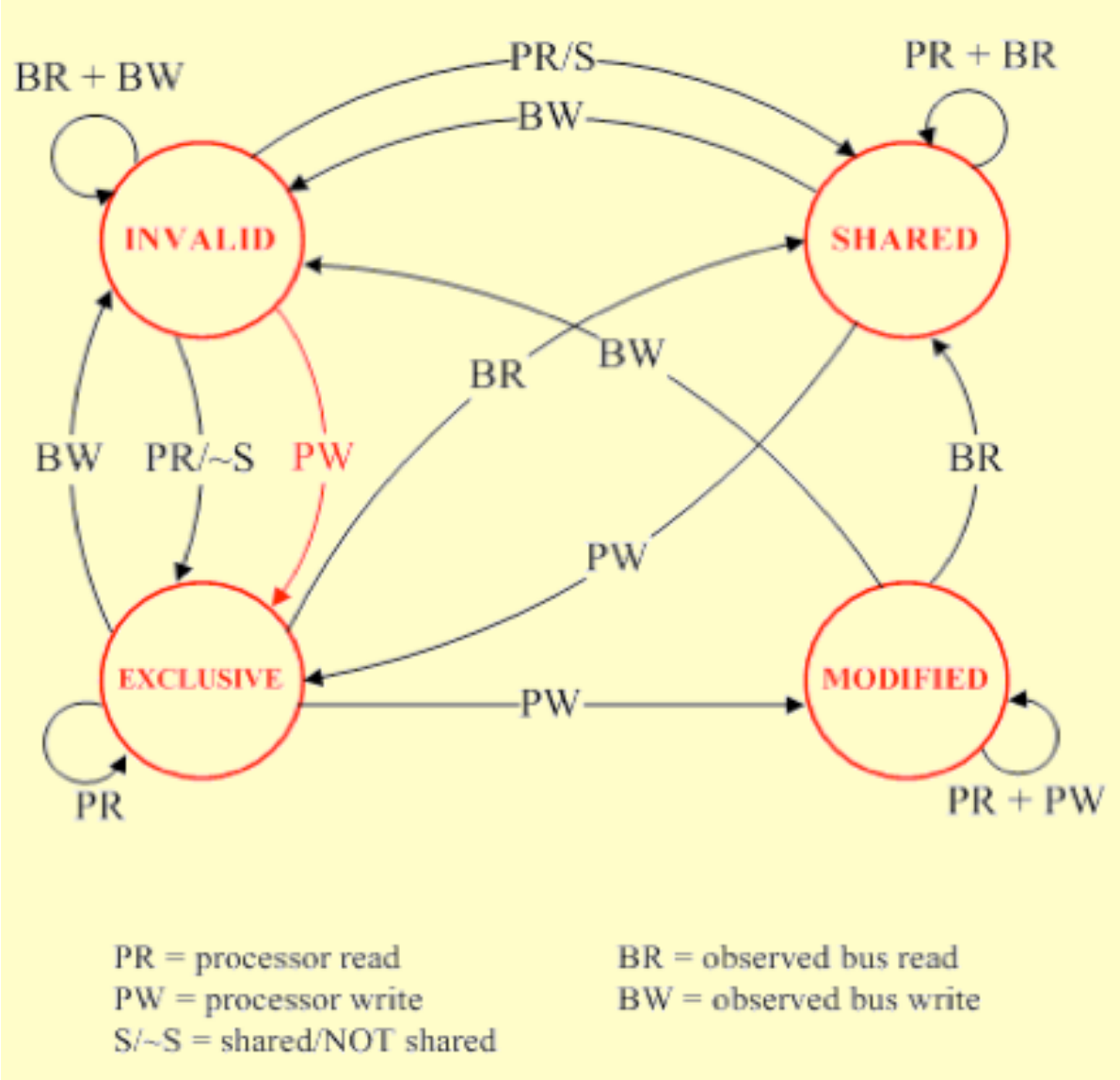
-> *Modified* upon write

Shared - clean and may be copied to another cache

-> *Invalid* upon other write

Invalid - dirty and may not be read/written/shared

MESI State Transition Diagram



PxQ

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

Memory Consistency Verification I: Memory Coherence

Memory Coherence: simply to specify, simple to verify for appropriately reduced model (formal) or appropriate sequential constraints (simulation).

Issues for Memory Coherence verification:

need to verify [WC] for every process P (modified to include synchronization - or at worst, verify [WDM] separately), relative to all the other processes; or [MC'], etc for other models.

Solution:

- replace "all other processes" by a generic process that acts like all the Q's together: a single generic Q should be enough (need to check this)
- verify for one P and appeal to symmetry to claim for all P's (may be few or many).

Easy ☺ !

But PV team wont believe it! They dont accept the goal. They want to verify the *cache coherency protocol*. (Cf. verifying arbiter spec vs implementation)

Memory Consistency Verification II: The Cache Coherence Protocol

The cache coherency protocol has many parts and many requirements, so this is HARD (and it doesnt even verify what you want - eg: P5 bug was in the protocol spec!).

Issues for cache coherency protocol verification:

- atomicity
- races
- deadlocks
- each MESI FSM works as it should
- reads, writes, write-backs work as they should

Can still appeal to symmetry, but this is a **lot of work** ☹ .

Squinting at the Future

- will Formal return to the foundry?
  - too much price pressure today on EDA
  - Growing impatience with EDA
  - BUT: don't forget the lesson of the 1990s: rolling your own is expensive
- Hierarchical Design
  - A methodological approach to system design that allows verification to scale
  - (Doesn't work everywhere)

Hierarchical Decomposition

- Design development today: **data before control**
  - Controllers need to point to defined data structures
  - But: upside down – often need to modify data structures for controllers
- Decompose vertically: **control before data**
  - Use stubs as place-holders for data
  - Controllers point to stubs
  - Stubs are oracles for data path computation
- Imposes hierarchical decomposition
  - Control at higher levels (coarse granularity supports global verification)
  - Data paths at lower levels (fine granularity verified locally)
  - Constant complexity at each level – scales with increasing design size

Hierarchical Decomposition, cont.

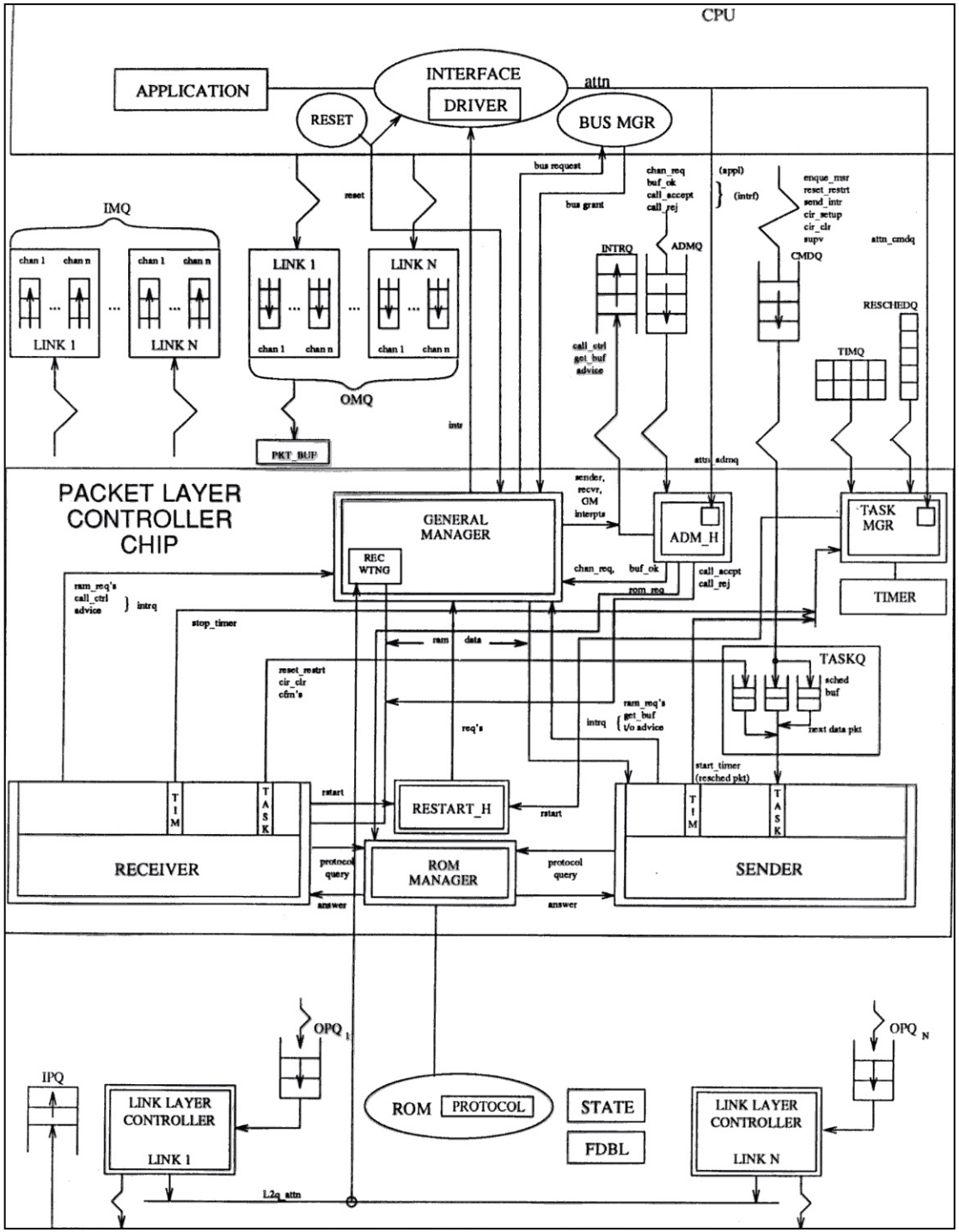
1. Start with functional spec, floor plan, etc
2. Derive properties (test plan) BEFORE coding design!
    1. Formal spec with comments (per Dijkstra!)
    2. Specification reviews (like design reviews) for completeness
3. Partition properties into levels
    1. Control properties first (global properties)
    2. Data path properties last (local properties)
4. Code to properties
    1. Use stubs as place-holders/oracles for lower levels
    2. Verify (simulation or formal) as you design

Implements top/down – bottom/up hierarchical design process

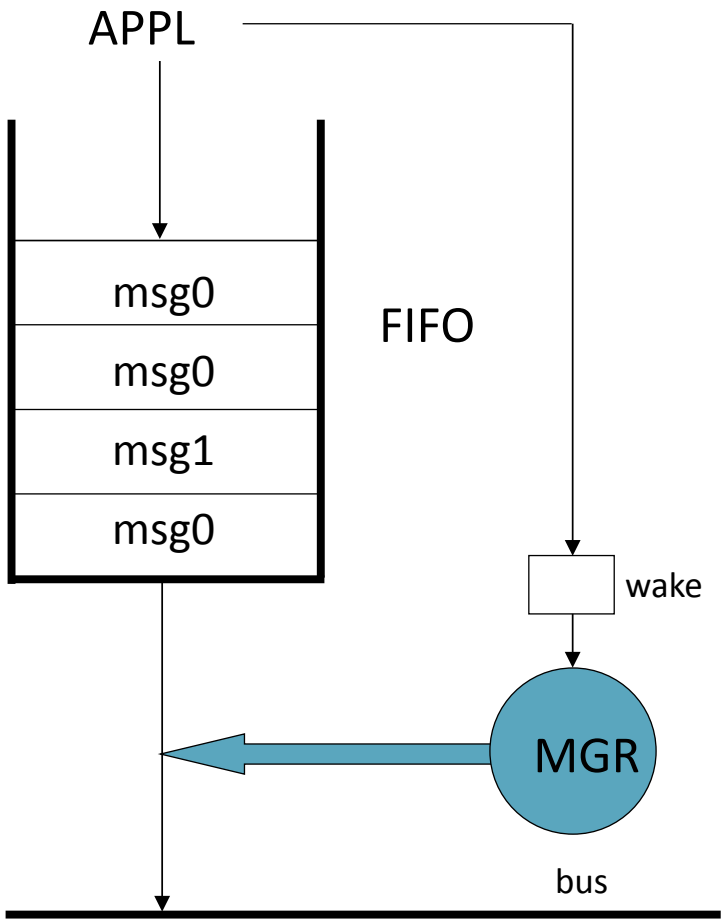
28 years ago …

Packet Layer Controller chip development at Bell Labs

- 200,000 transistors
- Developed entirely under the control of formal verification through a top/down stepwise refinement hierarchy
- 20% of projected cost
  6 staff years/2 calendar years vs projected 30 staff years
- "reliability of a 2nd generic release"


CPU
APPLICATION
INTERFACE
DRIVER
attn
RESET
BUS MGR
bus request
bus grant
chan_req
buf_ok
call_accept
call_rej
(appl)
(intrf)
enque_msr
reset_restrt
send_intr
cir_setup
cir_clr
supv
attn_cmdq
reset
IMQ
chan 1
chan n
LINK 1
LINK N
INTRQ
ADMQ
CMDQ
RESCHEDQ
TIMQ
call_ctrl
get_buf
advice
OMQ
intr
PKT_BUF
attn_admq
PACKET LAYER
CONTROLLER
CHIP
sender,
recvr,
GM
interpts
GENERAL
MANAGER
REC
WTNG
ADM_H
TASK
MGR
chan_req,
buf_ok
call_accept
call_rej
rom_req
TIMER
ram_req's
call_ctrl
advice
intrq
stop_timer
ram
data
TASKQ
sched
buf
reset_restrt
cir_clr
cfm's
ram_req's
get_buf
i/o advice
intrq
req's
next data pkt
start_timer
(resched pkt)
TIM
TASK
rstart
RESTART_H
RECEIVER
protocol
query
ROM
MANAGER
SENDER
answer
OPQ 1
OPQ N
IPQ
LINK LAYER
CONTROLLER
LINK 1
ROM
PROTOCOL
STATE
FDBL
LINK LAYER
CONTROLLER
LINK N
L2q_attn

Example: stubbing a FIFO – Lv1



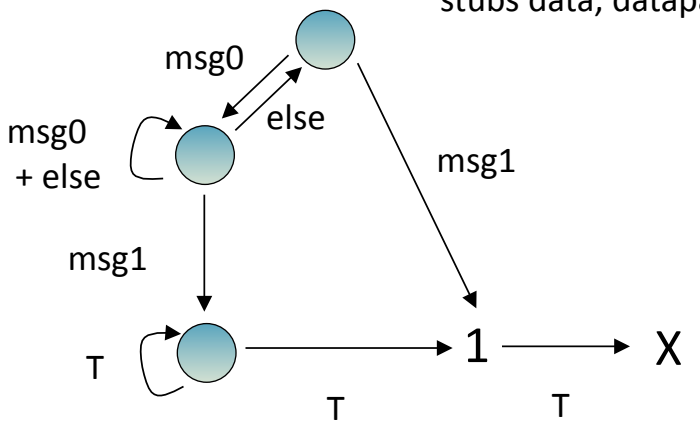
Lv1 data abstraction: track msg1,
All others -> msg0

Lv1 Assertion: After (*APPL.put_msg1*)
Eventually(*msg1_on_bus*)
[verifies MGR]

FIFO STUB

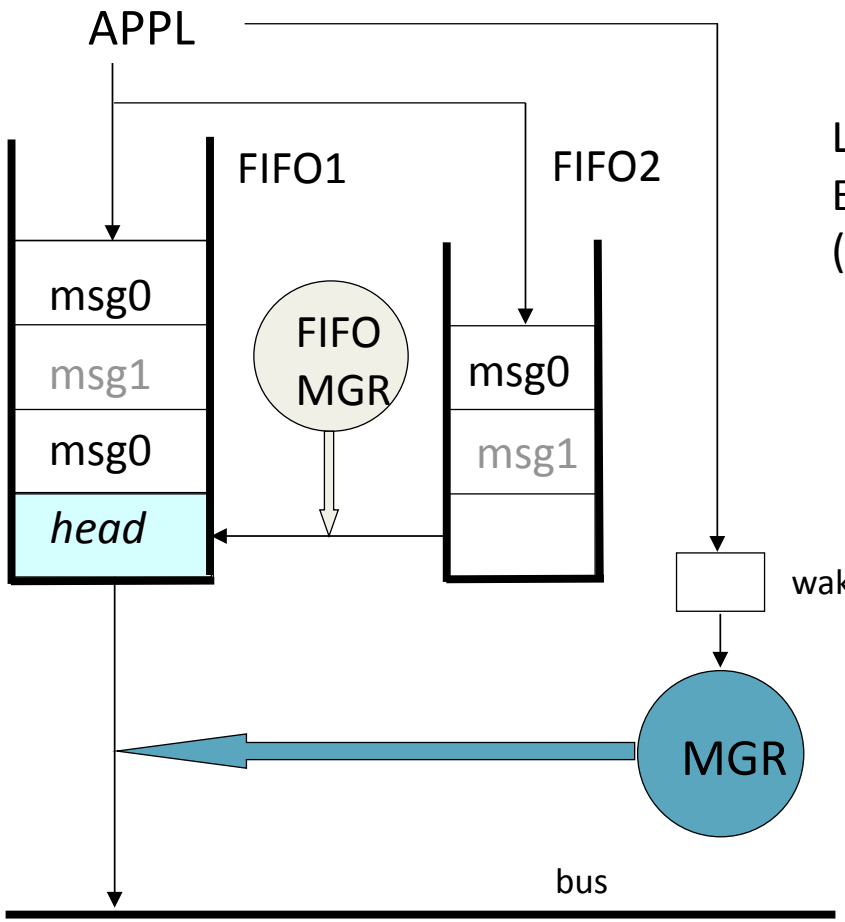
Lv1 Constraint: After (*FIFO.tail=msg1*)
Assume Eventually (*FIFO.head=msg1*)

stubs data, datapath



thanks Chris Komar

Example: stubbing a FIFO – Lv2



Lv2 refines FIFO stub into
2 sub-stubs
single msg1 can enter either

Lv2 Assertion: After (*FIFO.tail=msg1*)
Eventually (*FIFO.head=msg1*)
(= Lv1 constraint)
[Checks that FIFO MGR prevents starvation]

FIFO STUB

Lv2 Constraints:
After (*FIFO1.tail=msg1*)Assume Eventually (*FIFO1.head-1=msg1*)
After (*FIFO2.tail=msg1*)Assume Eventually (*FIFO2.head=msg1*)

thanks Vic Du

Further Refinements

Lv3: add FIFO mechanism (head/tail pointers)

- verify succession for real stages + abstract stage abstracting any number of words

  (verifies Lv2 constraints)

Lv4: expand abstract stage to full length of FIFO

- succession property follows inductively

Lv5: expand stages to full word width

- succession property follows inductively

Hierarchical Flow

Designer (D) ←→ Specification/Verification Engineer (S/VE)

work hand-in-hand

1. S/VE writes global (Lv1) assertions derived from Architectural Spec, block diagrams, floor plan, Functional Spec, …
2. D writes flow-control code to support testing of Lv1 assertions, writes stubs for associated lower-level (Lv2) structures
3. S/VE writes constraints on stubs (= Lv2* assertions)

* Hierarchy commonly not linear

4. D runs quick checks (limited/automatic model checking) of Lv1.

   -- debugs/fixes/iterates 4.

   -- S/VE works on DNFs: mc + reduction/decomp, hybrid, guided-random

Iterate 2, 3, 4.

5. D refines Lv1 stubs into Lv2 code to support testing of Lv2 assertions + Lv2 stubs (ie, 5. = 2. one level down)

Consequences

- Design and verification done together

  → earlier hence cheaper debug

  -- D sees bugs as they're encoded (not months later)

  -- debug when design is simpler, hence easier to fix (fewer adjacent consequences)

- PV promoted to S/VE
- D designs global flow control before low-level data structures (iteratively)

  →Designer focuses on function before structure

  -- structure serves function (today it's reverse)

  eg, requirements for memory coherence will precede and define requirements for a cache protocol (rather than reverse)

- Coverage/Capacity scales linearly with design size