Verifying cache coherence protocols for weak memory models

Chris Banks

THE UNIVERSITY *of* EDINBURGH
informatics

icsa | Institute for Computing Systems Architecture

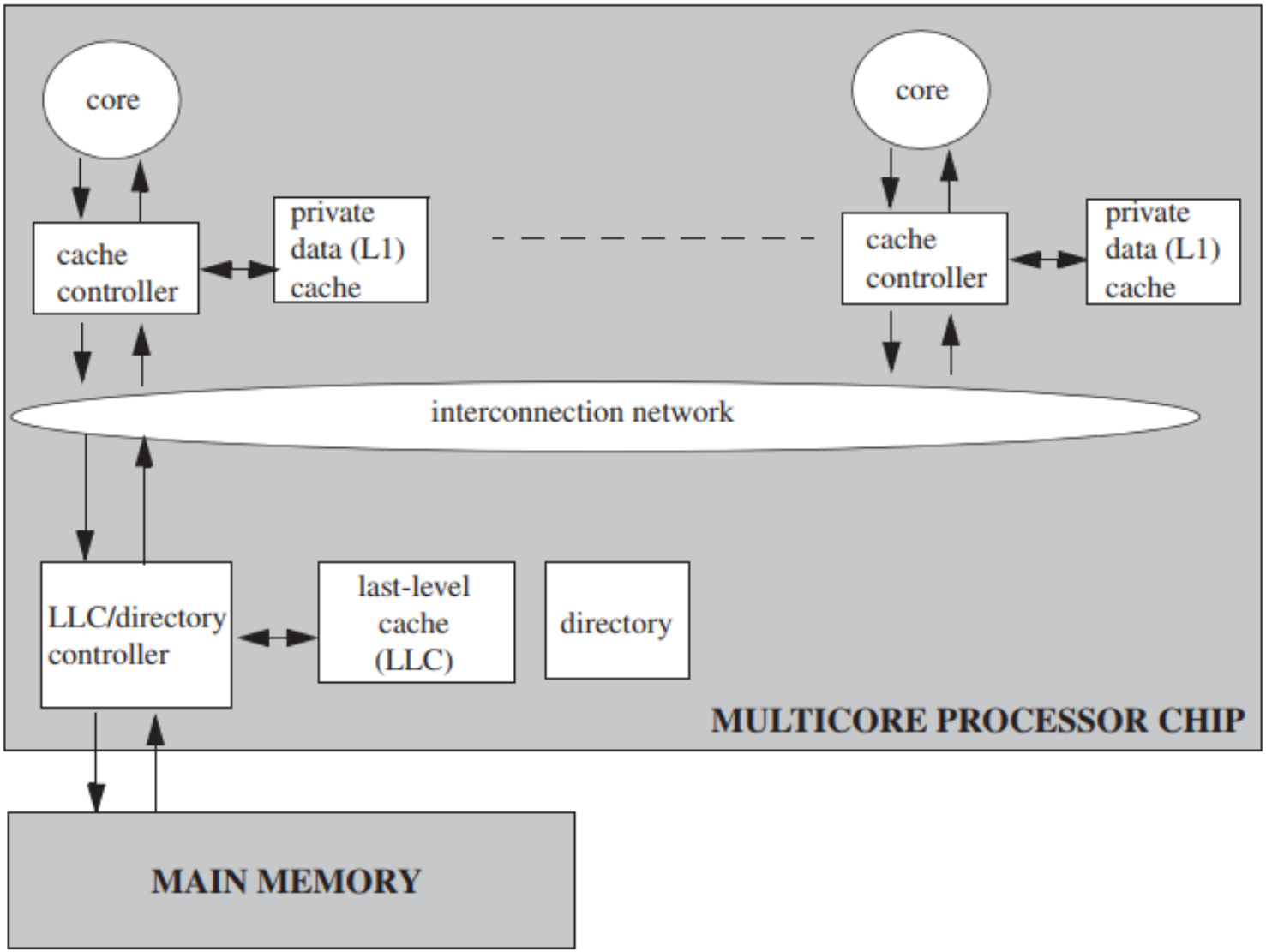
PEPA Club, 4th March 2016

Introduction

In this talk I will give:

- a whistle-stop tour of:
  - memory consistency models,
  - cache coherence protocols,
  - the relationship between the two;
- an overview of our project:
  - C3: Scalable and Verified Shared Memory via Consistency-directed Cache Coherence;
- demonstrate some of the verification work I've done already;
- and where it's going.

Multiprocessor architecture basics



1

[1] Sorin, Hill, & Wood: A Primer on Memory Consistency and Cache Coherence

Memory consistency model basics

In shared memory computer systems, memory consistency:

- defines shared memory correctness,
- provides rules about stores and loads, and how they act on memory,
- defines correct interleavings of concurrent operations,
- provides the *visible* programming model.

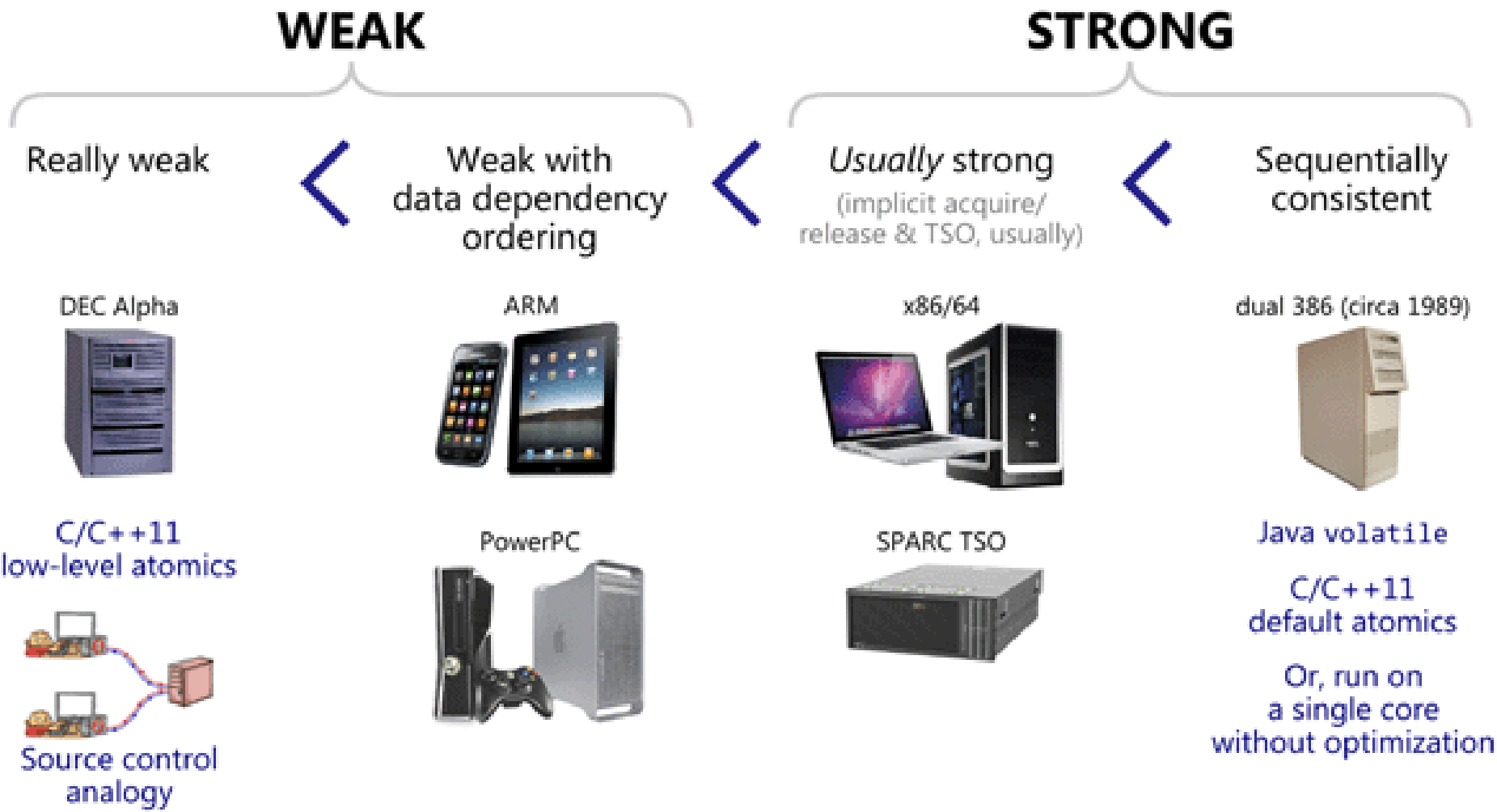
Cache coherence basics

In multiprocessors, the cache coherence protocol:

- defines the correct function of the cache system;
- aims to make caching *invisible* to the programmer:
  - *write propagation*: all cores eventually see all writes;
  - *write serialisation*: all cores see writes in the same order;
- ensures that caching does not change functionality.

Cache coherence protocols play a part in ensuring memory consistency.

Strong vs. Weak Consistency



2

[2] Figure: Jeff Preshing – http://preshing.com/20120930/weak-vs-strong-memory-models/

Sequential consistency

SC execution requires:

1. All cores insert loads/stores into memory (order $<_m$) in program order ($<_p$), regardless of addess ($a = b$ or $a \neq b$):
   - $L(a) <_p L(b) \implies L(a) <_m L(b)$ (Load $\rightarrow$ Load)
   - $L(a) <_p S(b) \implies L(a) <_m S(b)$ (Load $\rightarrow$ Store)
   - $S(a) <_p S(b) \implies S(a) <_m S(b)$ (Store $\rightarrow$ Store)
   - $S(a) <_p L(b) \implies S(a) <_m L(b)$ (Store $\rightarrow$ Load)
2. Every load gets its value from the last store before it (in $<_m$) to the same address:
   - $val(L(a)) = val(\max_{<_m}\{S(a) \mid S(a) <_m L(a)\})$

SC strictly enforces program ordering, so disallows the use of more optimised orderings.

Total Store Order

For example, SC can be relaxed to TSO (x86) to allow for *write buffering*:

1. All cores insert loads/stores into memory (order $<_m$) in program order ($<_p$), regardless of addess ($a = b$ or $a \neq b$):
   - $L(a) <_p L(b) \implies L(a) <_m L(b)$ (Load $\rightarrow$ Load)
   - $L(a) <_p S(b) \implies L(a) <_m S(b)$ (Load $\rightarrow$ Store)
   - $S(a) <_p S(b) \implies S(a) <_m S(b)$ (Store $\rightarrow$ Store)
   - ~~$S(a) <_p L(b) \implies S(a) <_m L(b)$~~ ~~(Store $\rightarrow$ Load)~~
2. Every load gets its value from the last store before it to the same address (in $<_m$ OR in $<_p$):
   - ~~$val(L(a)) = val(\max_{<_m}\{S(a) \mid S(a) <_m L(a)\})$~~
   - $val(L(a)) = val(\max_{<_m}\{S(a) \mid S(a) <_m L(a) \vee S(a) <_p L(a)\})$

Total Store Order

The programmer (or compiler) must prevent unwanted execution orders by inserting fences (or synchronisation barriers).

So we add the FENCE rules:

3. Fences order everything:
   - $S(a) <_p F \implies S(a) <_m F$ (Store $\rightarrow$ FENCE)
   - $F <_p L(a) \implies F <_m L(a)$ (FENCE $\rightarrow$ Load)
   - Other FENCE orderings are enforced by Store $\rightarrow$ Load.

Cache Coherence Model

Classical cache coherence protocols generally enforce the following:

1. Single-Writer, Multiple-Reader (SWMR):
   - for any memory location, at any (logical) time, there exists only one core that may write to the memory location;
   - or any number of cores that may only read from it;
   - enforces *write serialisation*.
2. Data-value invariant:
   - Define RO/RW *epochs* of a given memory location, e.g.:

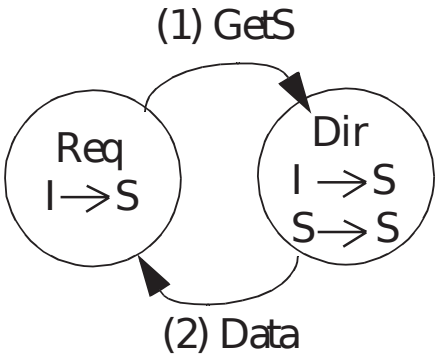
     RO(C1,C2) ; RW(C3) ; RW(C1) ; RO(C1,C2,C3) ; ...
   - The value of a location at the start of an epoch is the same as at the end of its last read-write epoch.
3. Write propagation: by eagerly invalidating copies in other processors.

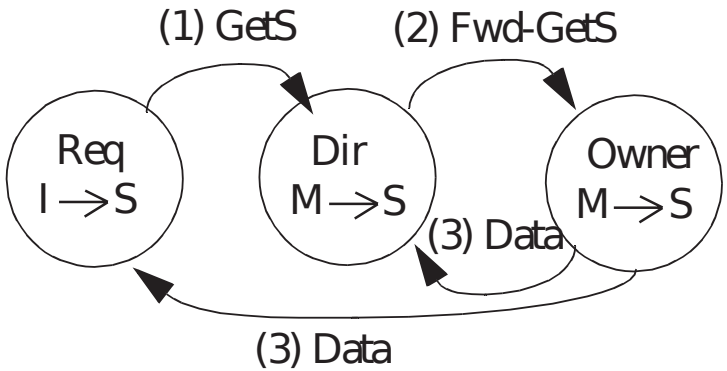
A very simple CCP

A very simple CCP:

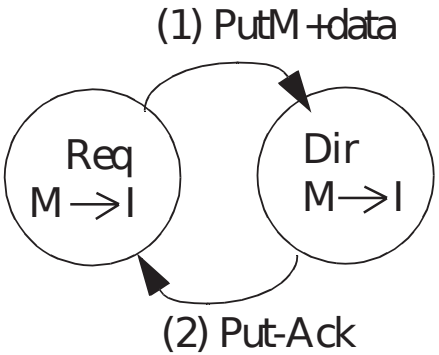
- Uses a directory to keep track of cache states;
- Cache states for each memory location can be:
  - **M**odified – owned by a single core for writing;
  - **S**hared – has a current value, shared by one or more cores for reading;
  - **I**nvalid – has a potentially stale value or no value.

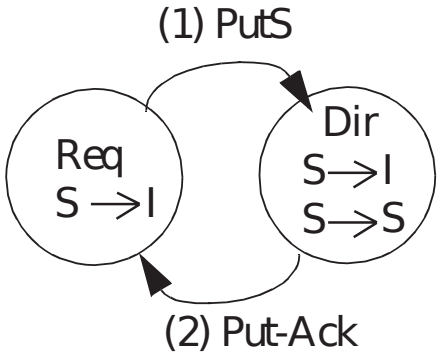
MSI Directory Protocol





Transitions from I to S.



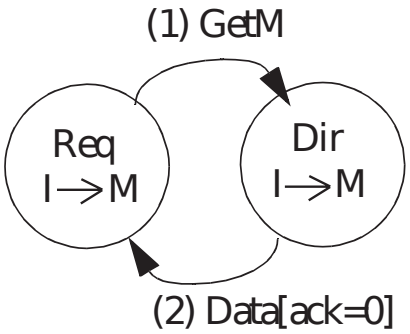


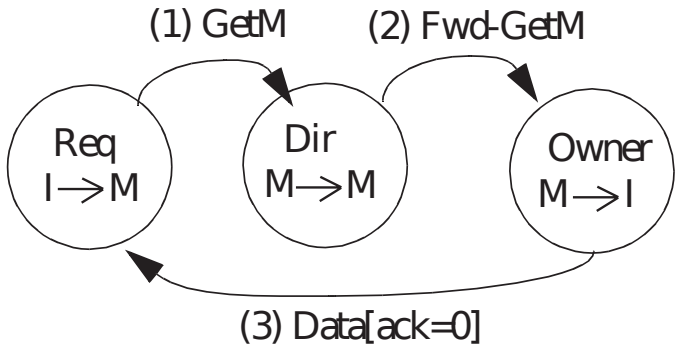
Transition from M or S to I

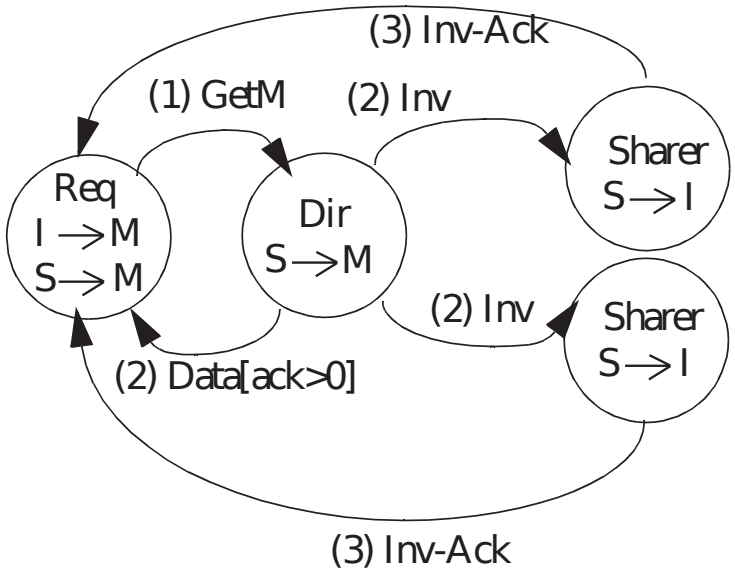
[3]

[3] Sorin, Hill, & Wood: A Primer on Memory Consistency and Cache Coherence

MSI Directory Protocol







The only sharer might be the requestor, in which case no Invalidation messages are sent and the Data message from the Dir to Req has an AckCount of zero.

Transitions from I or S to M

4

[4] Sorin, Hill, & Wood: A Primer on Memory Consistency and Cache Coherence

MSI Directory Protocol

TABLE 8.2: MSI Directory Protocol—Directory Controller

	GetS	GetM	PutS-NotLast	PutS-Last	PutM+data from Owner	PutM+data from NonOwner	Data
I	send data to Req, add Req to Sharers/S	send data to Req, set Owner to Req/M	send Put-Ack to Req	send Put-Ack to Req		send Put-Ack to Req	
S	send data to Req, add Req to Sharers	send data to Req, send Inv to Sharers, clear Sharers, set Owner to Req/M	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req/I		remove Req from Sharers, send Put-Ack to Req	
M	Send Fwd-GetS to Owner, add Req and Owner to Sharers, clear Owner/S^D	Send Fwd-GetM to Owner, set Owner to Req	send Put-Ack to Req	send Put-Ack to Req	copy data to memory, clear Owner, send Put-Ack to Req/I	send Put-Ack to Req	
S^D	stall	stall	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req		remove Req from Sharers, send Put-Ack to Req	copy data to memory/S

[5]

[5] Sorin, Hill, & Wood: A Primer on Memory Consistency and Cache Coherence

MSI Directory Protocol

TABLE 8.1: MSI Directory Protocol—Cache Controller

	load	store	replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from Owner	Inv-Ack	Last-Inv-Ack
I	send GetS to Dir/IS^D	send GetM to Dir/IM^{AD}										
IS^D	stall	stall	stall			stall		-/S		-/S		
IM^{AD}	stall	stall	stall	stall	stall			-/M	-/IM^A	-/M	ack--	
IM^A	stall	stall	stall	stall	stall						ack--	-/M
S	hit	send GetM to Dir/SM^{AD}	send PutS to Dir/SI^A			send Inv-Ack to Req/I						
SM^{AD}	hit	stall	stall	stall	stall	send Inv-Ack to Req/IM^{AD}		-/M	-/SM^A	-/M	ack--	
SM^A	hit	stall	stall	stall	stall						ack--	-/M
M	hit	hit	send PutM+data to Dir/MI^A	send data to Req and Dir/S	send data to Req/I							
MI^A	stall	stall	stall	send data to Req and Dir/SI^A	send data to Req/II^A		-/I					
SI^A	stall	stall	stall			send Inv-Ack to Req/II^A	-/I					
II^A	stall	stall	stall				-/I					

[6]

[6] Sorin, Hill, & Wood: A Primer on Memory Consistency and Cache Coherence

More relaxed coherence

- Classical protocols don't take any advantage of relaxed memory models.
- More modern protocols don't eagerly enforce coherence on every write—only at synchronisation boundaries.
- Shared lines self-invalidate at synchronisation boundaries—consistency enforced lazily (in logical time).
- Several approaches to limit self-invalidations (increased L1 misses), e.g. via timestamps in TSO-CC (Elver and Nagarajan, 2014).

The C3 Project

C3: Scalable & Verified Shared Memory via Consistency-directed Cache Coherence

- The decoupling of MCMs and CCPs means:
  - CCPs are not well-optimised for relaxed MCMs;
  - verification and scalability has been compromised.
- Even modern protocols don't scale well to large numbers of cores.
- Whilst protocols may be verified in isolation, they're rarely verified against the MCM—bugs have been found.
- **The aim is to design efficient lazy CCPs, verified against relaxed MCMs.**

So far...

- Investigating the limits of the current (erm... since the 90s) approach.
- Verification of MSI-Dir protocol against SC.
- Murϕ model checker.

[Demo]

Next steps

- Immediately:
  - Verification by simulation with operational model of MCM.
  - Verification of basic TSO-CC, lazy coherence protocol (Marco Elver).
- Then:
  - Tackle state space explosion—parameterised techniques?
  - How to verify TSO-CC with timestamps?
  - (Some timestamped crypto protocols are verified using Timed Automata.)
- Future:
  - Synthesis of verifiable protocols?
  - Formal language and tools for specifying and verifying CCPs?

Fin

Questions?