McVerSi: A Test Generation Framework for Fast Memory Consistency Verification in Simulation

Marco Elver Vijay Nagarajan

University of Edinburgh

HPCA, March 2016

Overview

Problem

- How to rigorously verify hardware implementation (full system functional design) satisfies memory consistency model in *simulation*?
- Simulation-based verification slow: takes long to achieve high coverage $\rightarrow$ low assurance.

McVerSi: Evolutionary Test Generation Approach (GP)

- Carefully recombines tests, favoring racy operations.
- Coverage directed.

Results

- Bugs found faster and more consistent than random test generation and litmus tests.
- Found all tested bugs in less than 1 day, unlike alternatives.
- Found 2 new bugs in Gem5.

Motivation

Memory consistency verification still a problem

- Real silicon still ships with consistency bugs[1] [2].
- Major challenges:
  1. In testing based approaches: how to cover all relevant aspects of implementation?
  2. In formal verification: does translated abstraction represent real implementation sufficiently?
  3. Which components (regardless of method): can miss bugs due to interaction if not modelled correctly!
- Want rigorous verification of functional design implementation, e.g. *cycle accurate model* of full system.

[1] Alglave et al., “Herding cats”, In: TOPLAS, 2014.

[2] Alglave et al., “GPU concurrency: Weak behaviours and programming assumptions”, In: ASPLOS, 2015.

Memory Consistency Models

Consistency: contract between programmer and hardware

$\Rightarrow$ Specifies how memory system should behave: conveys ordering guarantees (or lack of) among memory ops.

Some Common MCMs

- SC: all program orders maintained.
- TSO: relaxes *write* $\rightarrow$ *read* order.

init: x = 0, y = 0	
Thread 1	**Thread 2**
x $\leftarrow$ 1	r1 $\leftarrow$ y
y $\leftarrow$ 1	r2 $\leftarrow$ x
Allow?: r1 = 1 $\wedge$ r2 = 0	

Memory Consistency Models

Consistency: contract between programmer and hardware

$\Rightarrow$ Specifies how memory system should behave: conveys ordering guarantees (or lack of) among memory ops.

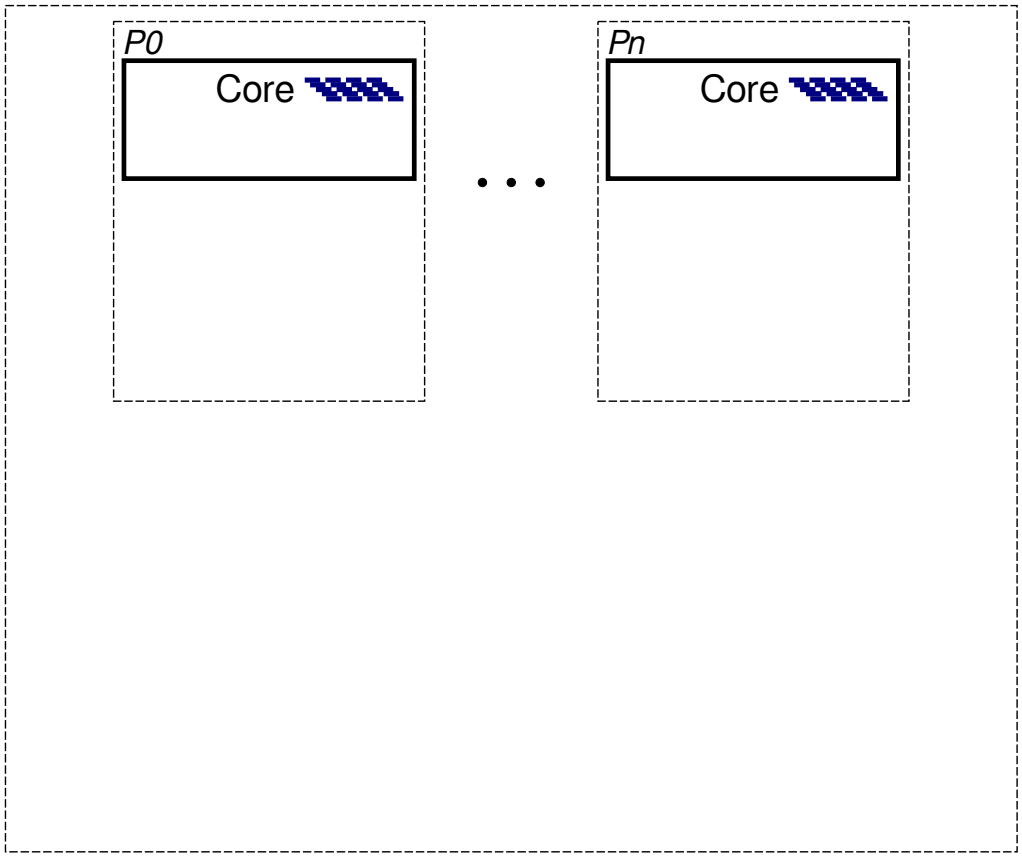
Some Common MCMs

- SC: all program orders maintained.
- TSO: relaxes *write* $\rightarrow$ *read* order.

init: x = 0, y = 0	
Thread 1	**Thread 2**
x $\leftarrow$ 1	y $\leftarrow$ 1
r1 $\leftarrow$ y	r2 $\leftarrow$ x
Allow?: r1 = 0 $\wedge$ r2 = 0	

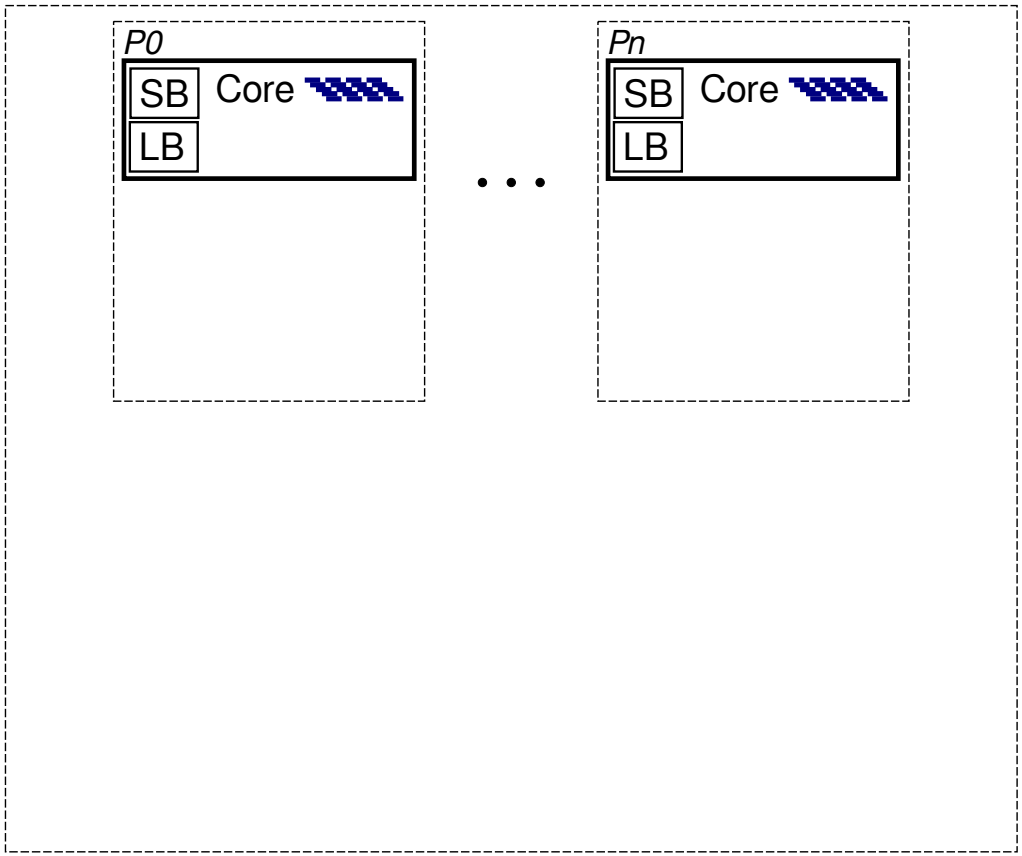
What components can affect the MCM?

- Core pipeline (in-order, OoO).



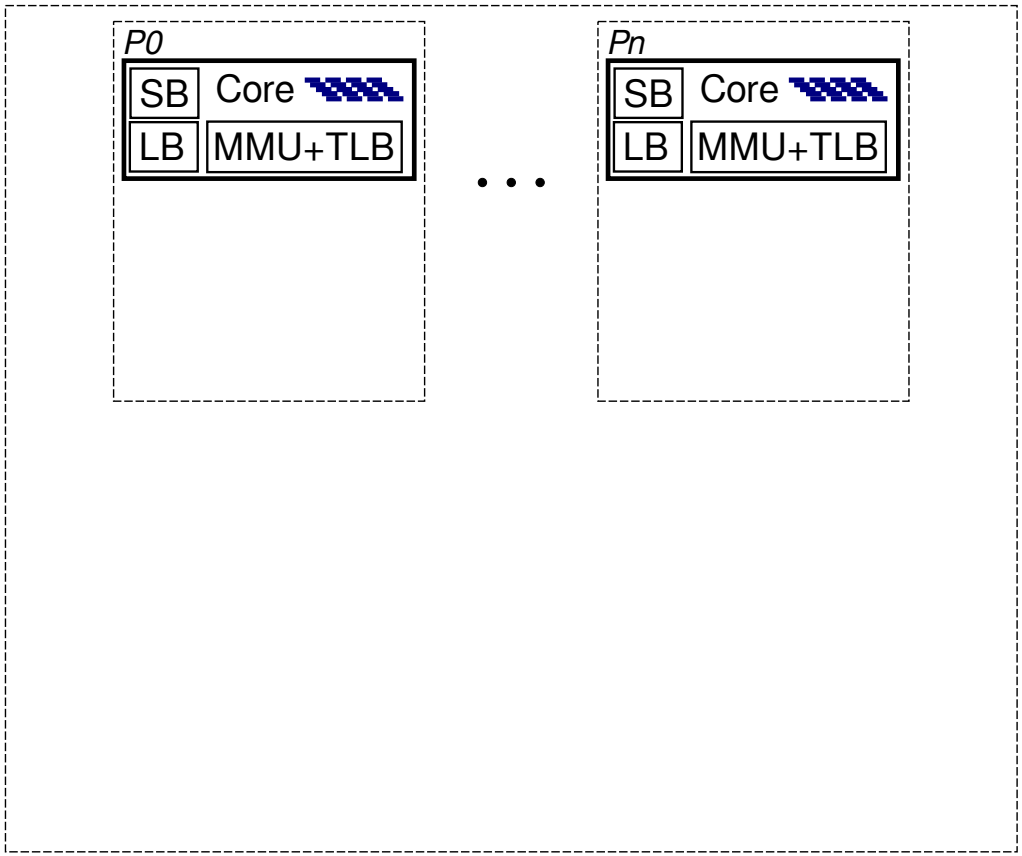
What components can affect the MCM?

- Core pipeline (in-order, OoO).



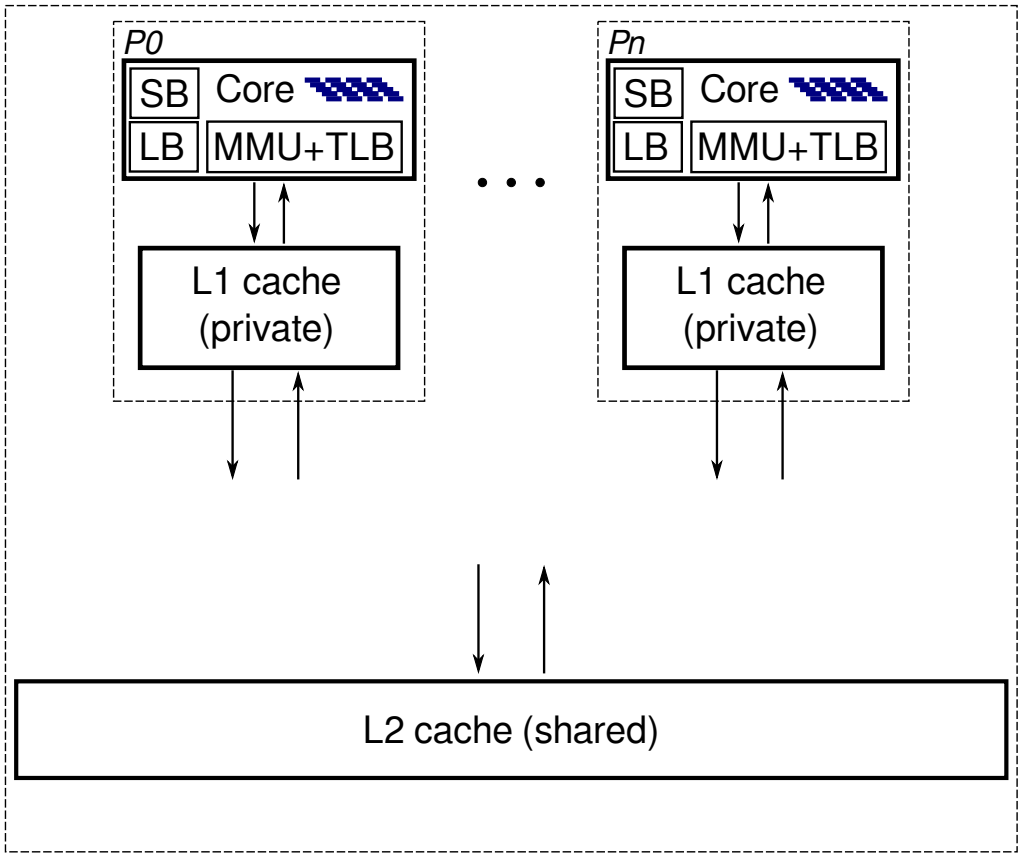
What components can affect the MCM?

- Core pipeline (in-order, OoO).
- MMU+TLB.



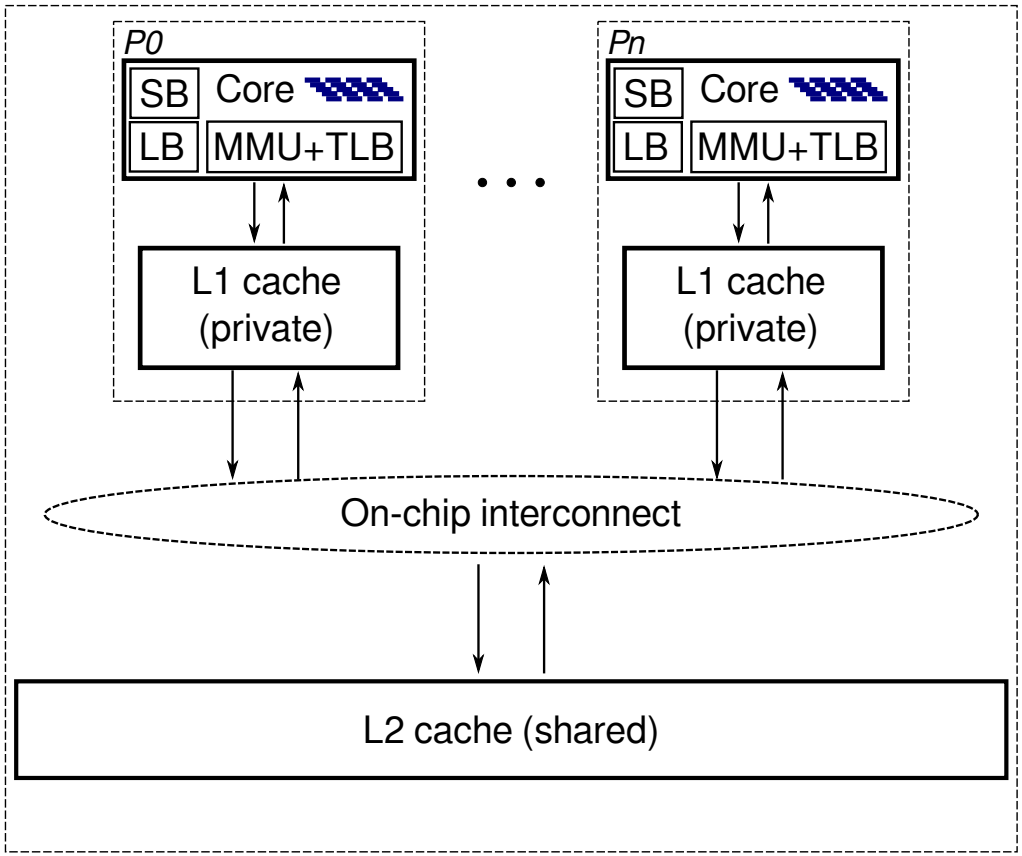
What components can affect the MCM?

- Core pipeline (in-order, OoO).
- MMU+TLB.
- Caches + coherence protocol (eager, lazy).



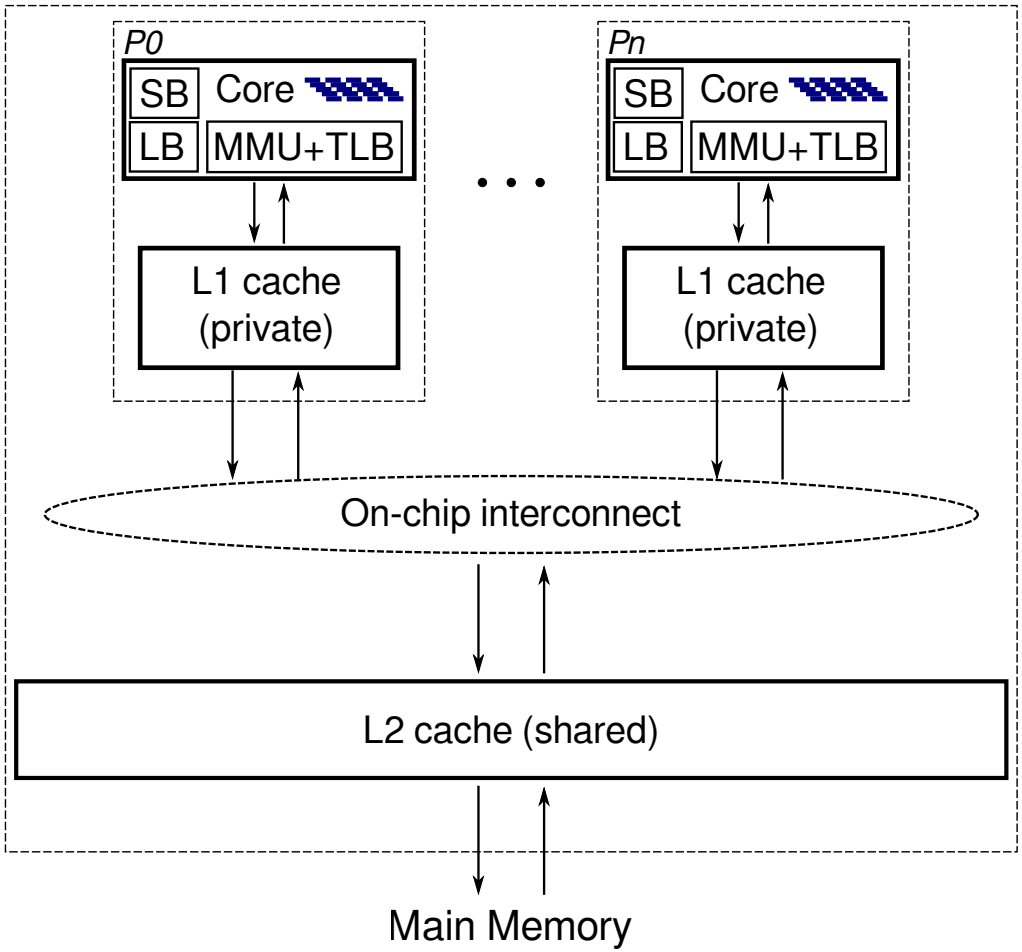
What components can affect the MCM?

- Core pipeline (in-order, OoO).
- MMU+TLB.
- Caches + coherence protocol (eager, lazy).
- Interconnect.



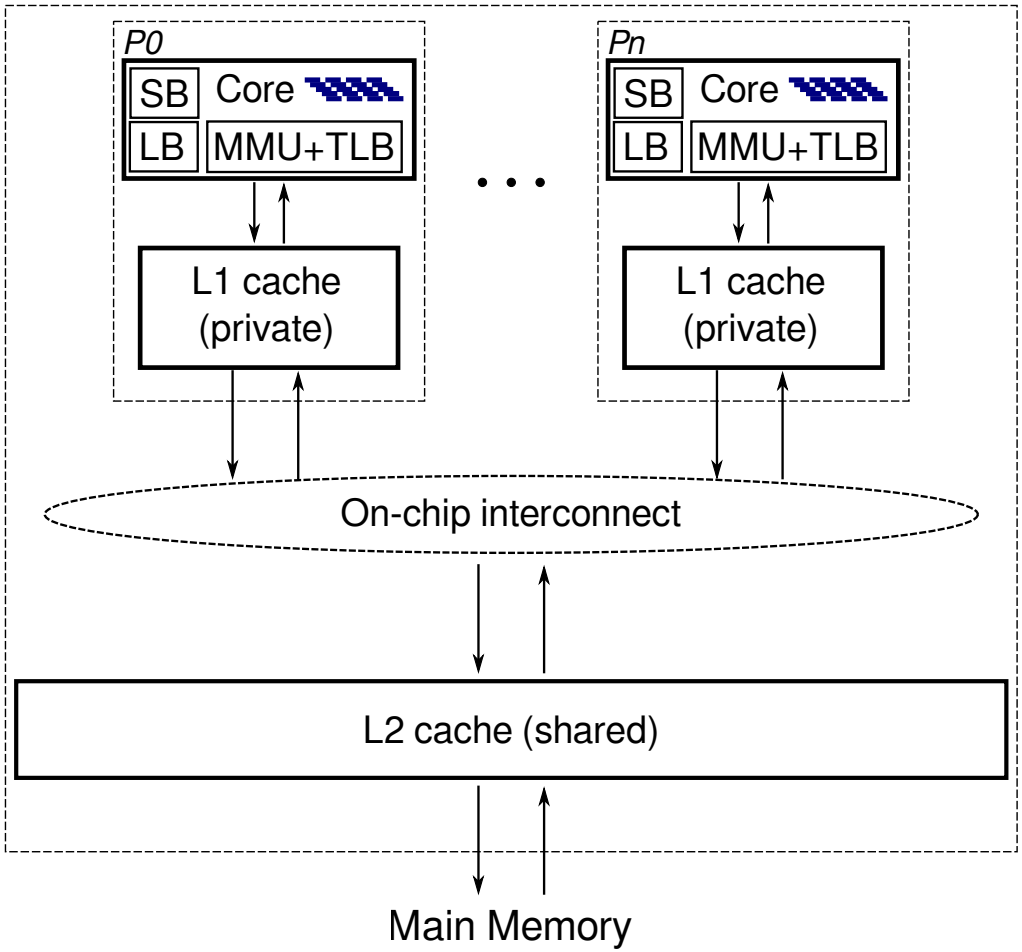
What components can affect the MCM?

- Core pipeline (in-order, OoO).
- MMU+TLB.
- Caches + coherence protocol (eager, lazy).
- Interconnect.
- ...



What components can affect the MCM?

- Core pipeline (in-order, OoO).
- MMU+TLB.
- Caches + coherence protocol (eager, lazy).
- Interconnect.
- ...



Composition+interaction of components must still satisfy MCM!

How to verify?

Formal verification

- Provides proof; e.g. model checking (exhaustive search) [Dill, 1996].
- Works for high-level abstractions, individual components.
- Impractical for detailed (e.g. cycle accurate) full system design.

Post-silicon verification

- Random testing (e.g. TSOtool [Hangal et al., 2004]), etc.
- Fastest $\rightarrow$ good coverage (high assurance).
- Expensive; hard to debug.

How to verify?

Formal verification

- Provides proof; e.g. model checking (exhaustive search) [Dill, 1996].
- Works for high-level abstractions, individual components.
- Impractical for detailed (e.g. cycle accurate) full system design.

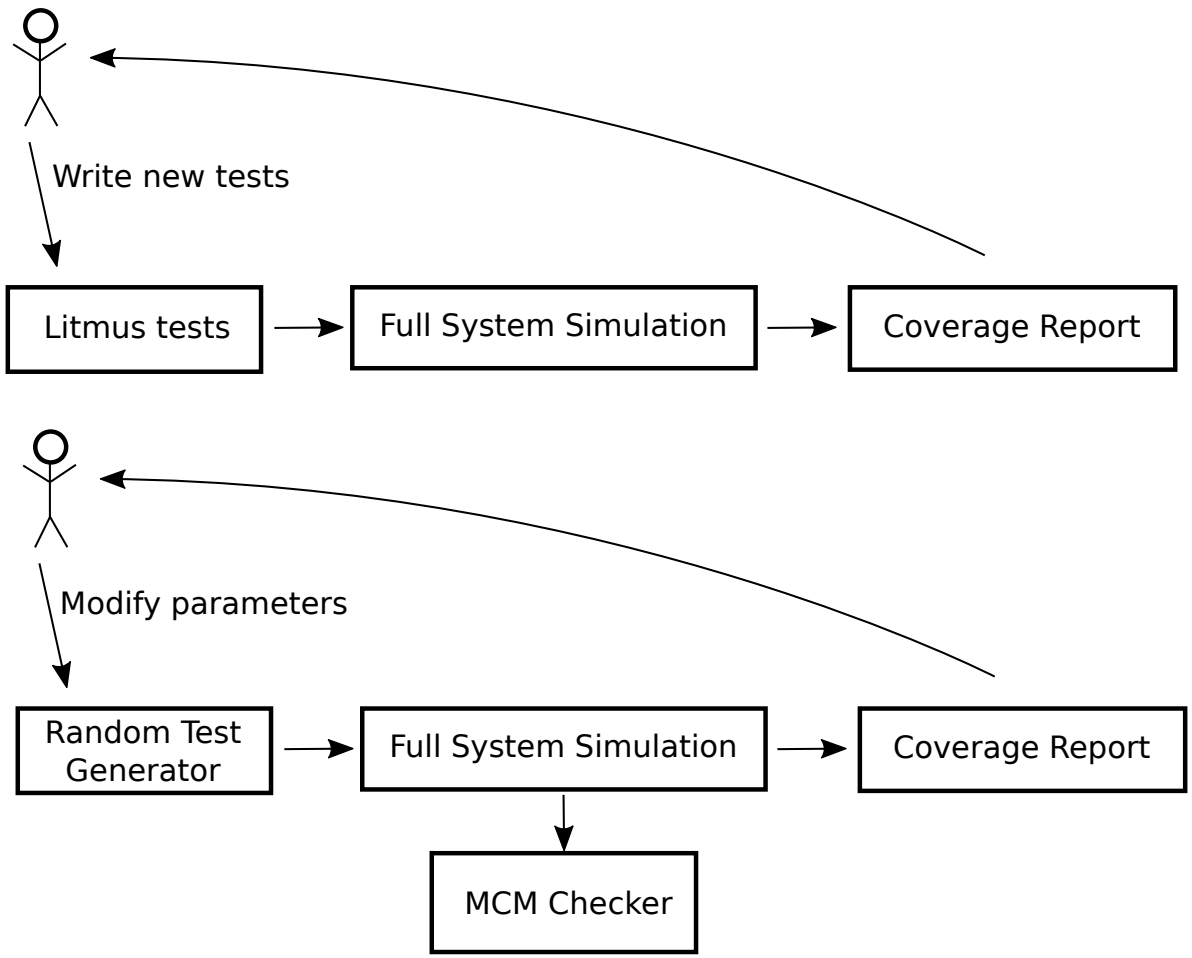
Simulation-based verification of full system

- Litmus tests, pseudo-random tests [Saha et al., 1995].
- Cheap; easy to debug.
- Extremely slow $\rightarrow$ poor coverage (low assurance).

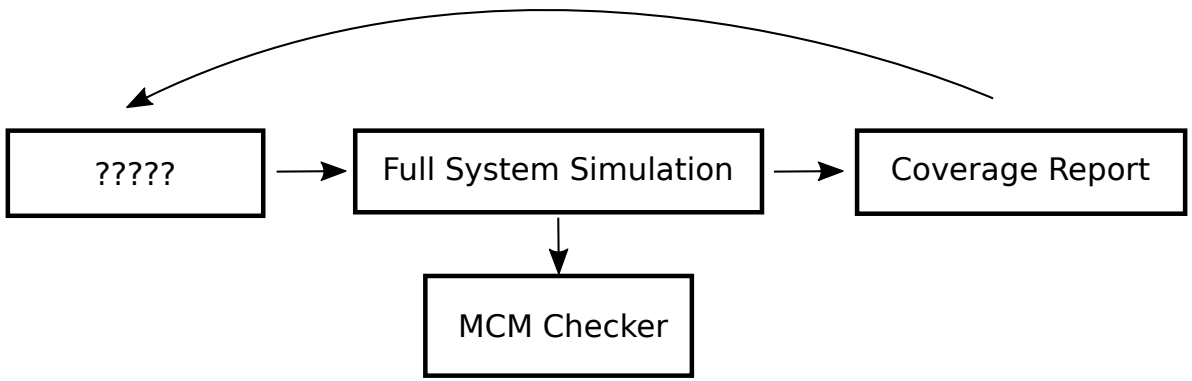
Post-silicon verification

- Random testing (e.g. TSOtool [Hangal et al., 2004]), etc.
- Fastest $\rightarrow$ good coverage (high assurance).
- Expensive; hard to debug.

How to verify in simulation?



How to automate?

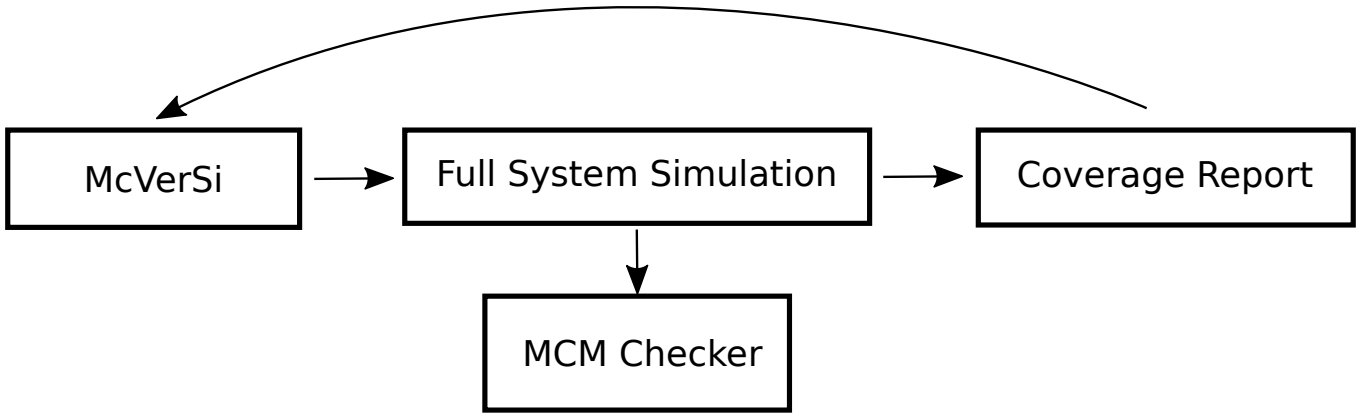


Problem Statement

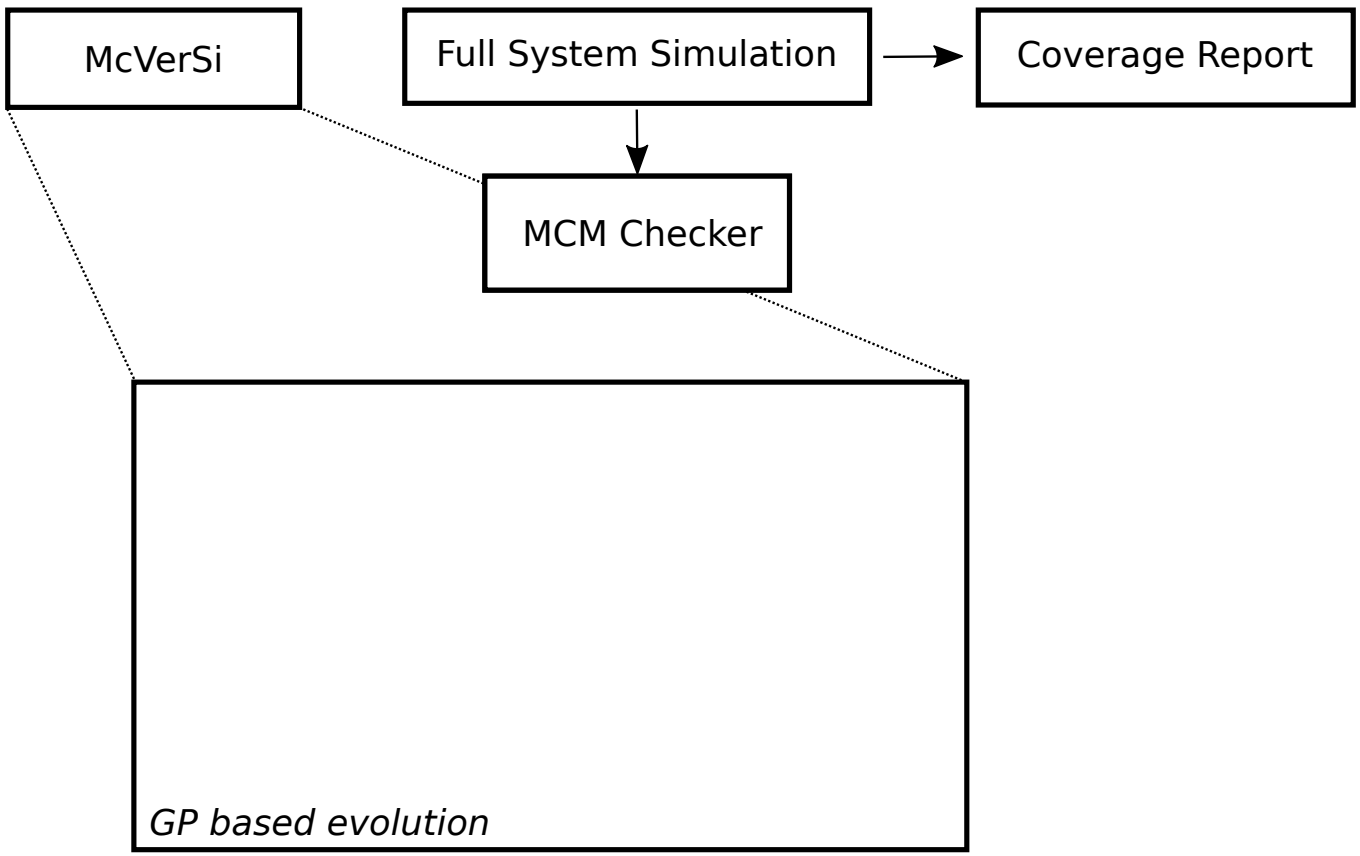
How to generate efficient MCM tests for simulation, s.t. wall-clock time to explore rare corner cases is reduced?

1. What are good MCM tests?
2. How to feed back coverage into test generator?

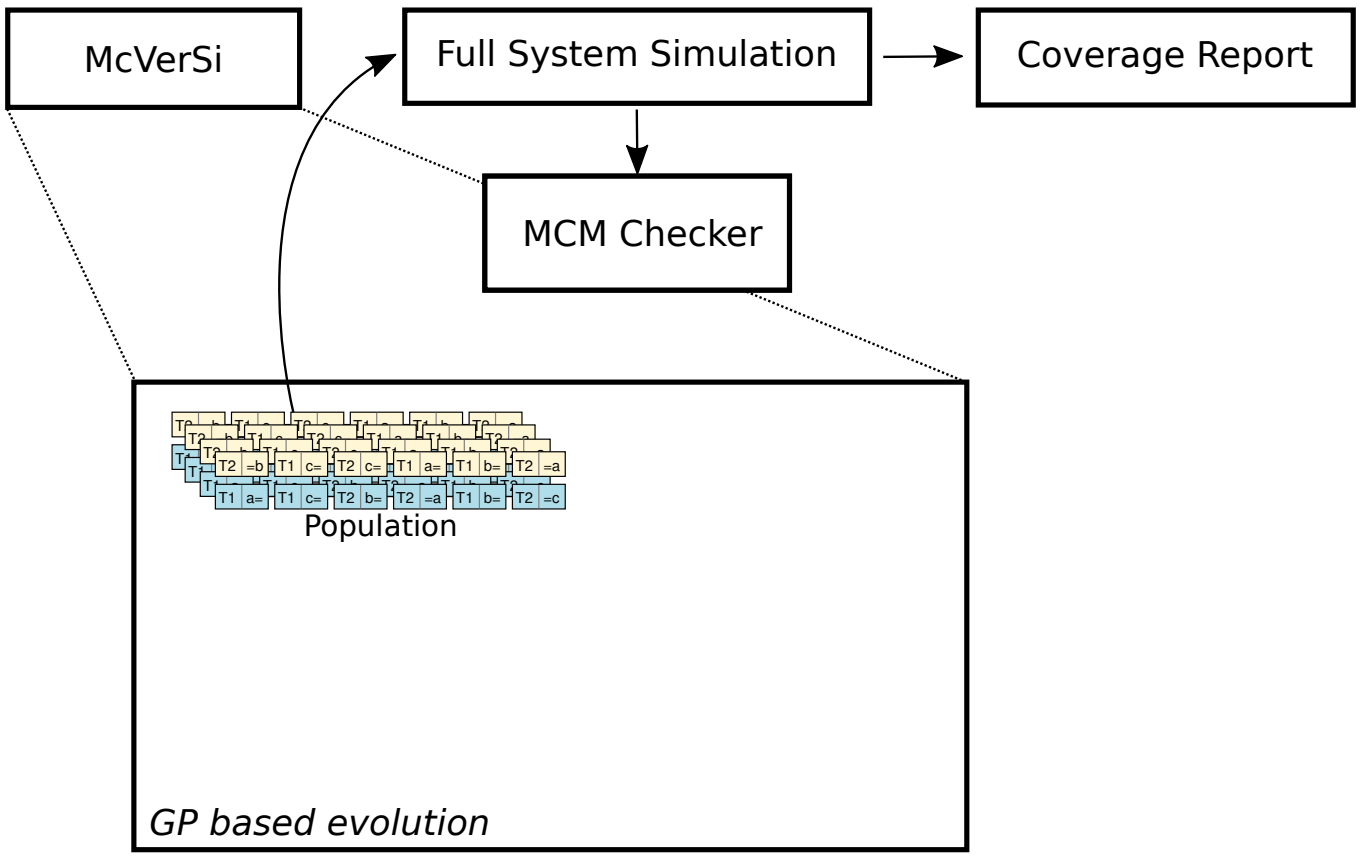
McVerSi Approach



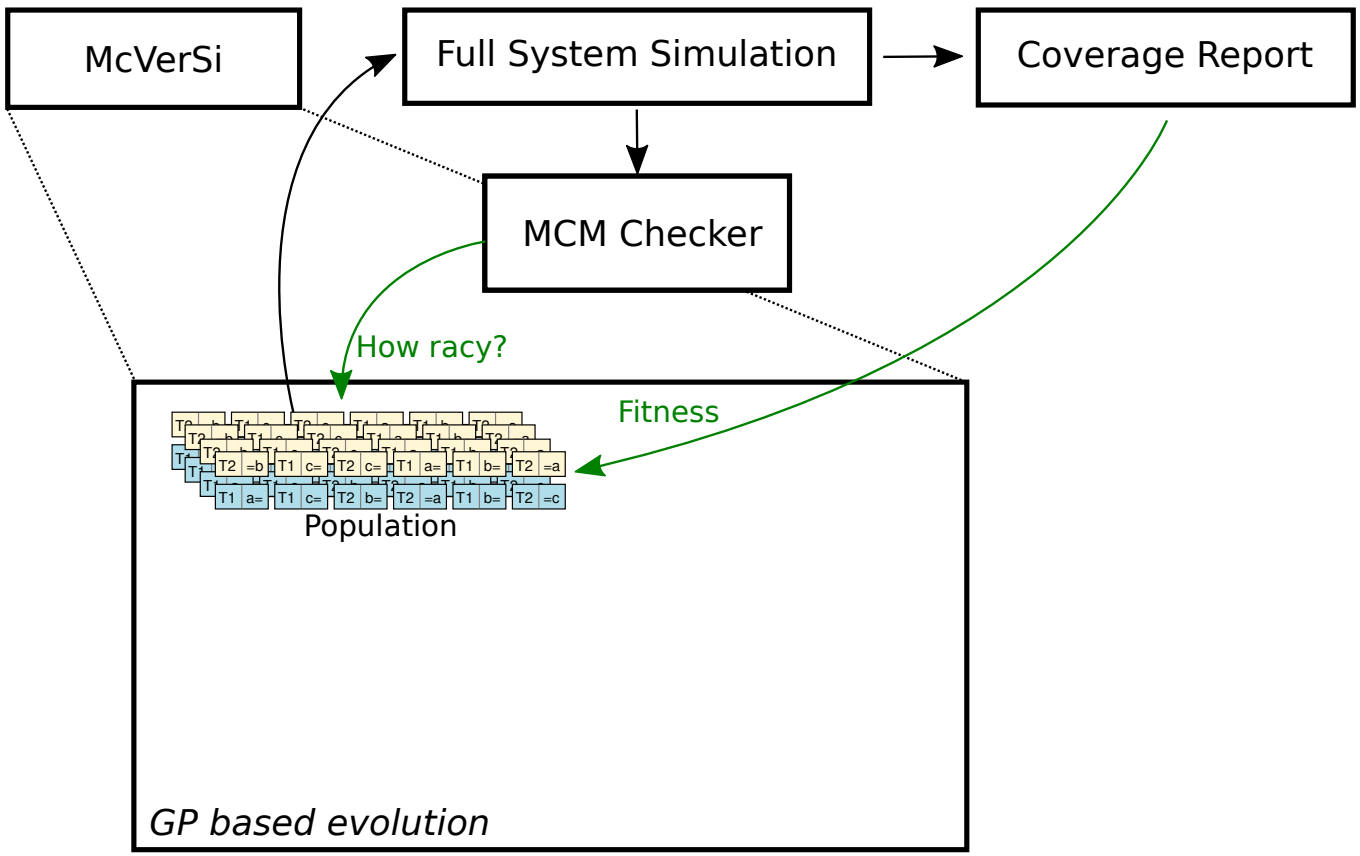
McVerSi Approach



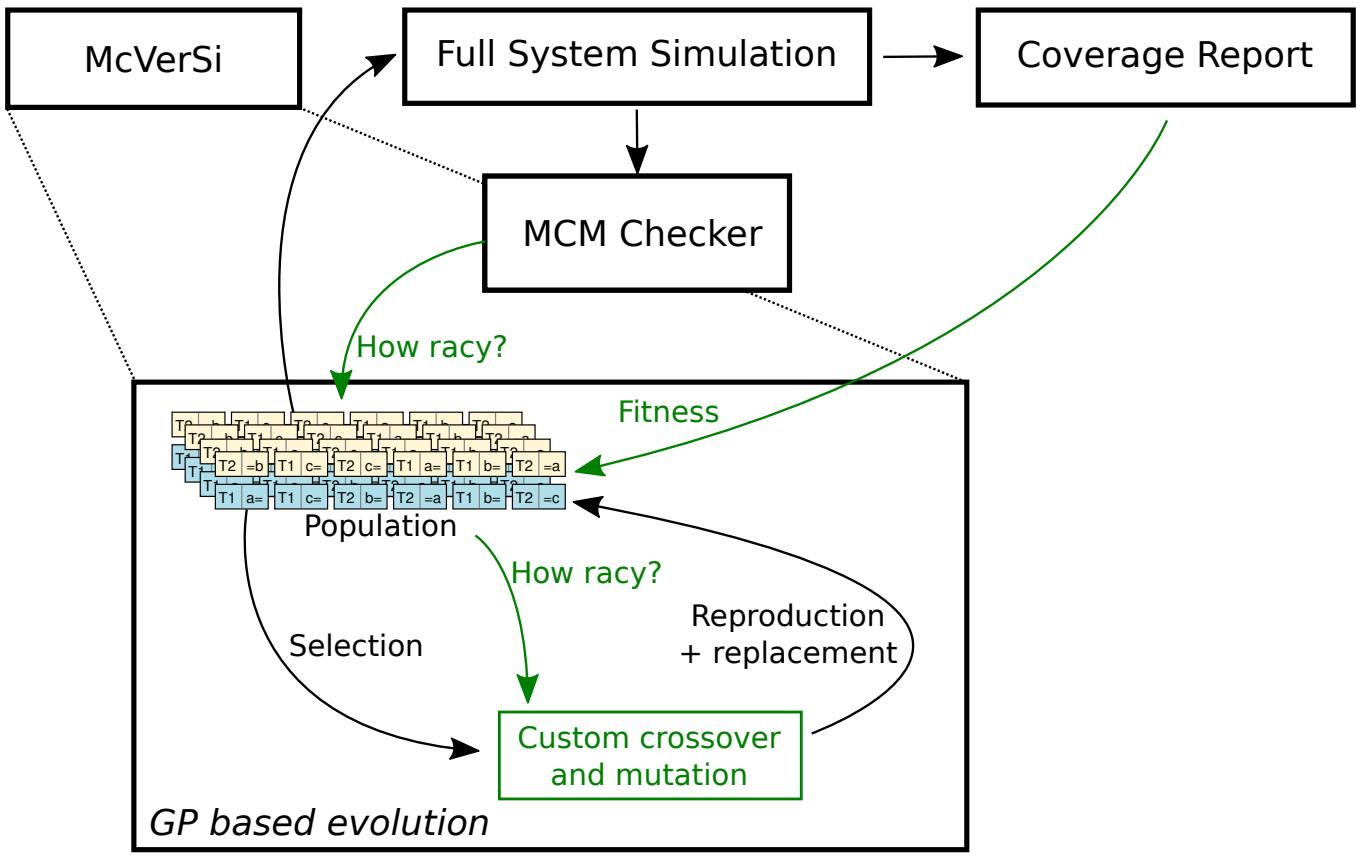
McVerSi Approach



McVerSi Approach

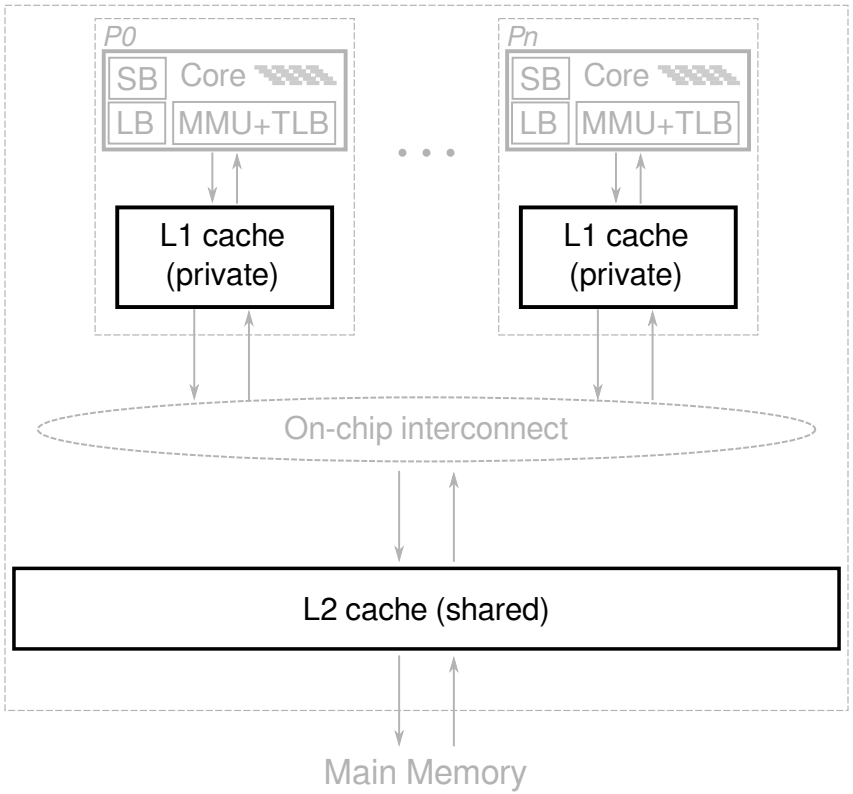


McVerSi Approach



Measuring Fitness/Coverage

- Simulation runs continuously, with new tests loaded on-the-fly.
- Test fitness: result of coverage of individual *test-run*.
- Evaluated with coherence protocol transition coverage.



Measuring Fitness/Coverage

- Simulation runs continuously, with new tests loaded on-the-fly.
- Test fitness: result of coverage of individual *test-run*.
- Evaluated with coherence protocol transition coverage.

One of many possible options—highly dependent on design!

Test Suitability: What is a good MCM test?

init: x = 0, y = 0	
Thread 1	**Thread 2**
x ← 1	r1 ← y
y ← 1	r2 ← x
r1 = {0, 1} ∧ r2 = {0, 1}	

Average non-determinism of test (ND_T)

Average number of races observed per memory operation (ND_e) across all iterations of a test-run → higher implies better test.

Test Suitability: What is a good MCM test?

init: x = 0, y = 0, z = 0	
Thread 1	**Thread 2**
$z \leftarrow 1$	$r1 \leftarrow y$
$y \leftarrow 1$	$r2 \leftarrow x$
$r1 = \{0, 1\} \wedge r2 = \{0\}$	

Average non-determinism of test (ND_T)

Average number of races observed per memory operation (ND_e) across all iterations of a test-run $\rightarrow$ higher implies better test.

Selective Crossover

Crossover

- Crossover operator crucial in GP to create new program(s) from 2 parents with good fitness: offspring likely to outperform (in fitness).

Goal: Encode Domain Knowledge

- Use knowledge of what makes good MCM test.

Crossover Algorithm

1. Obtain *fitaddrs*: addresses of operations where ND_e is larger than rounded ND_T of test.
2. Select all operations with address in *fitaddrs* (discard others).
3. Overlay both parent tests, resulting in new test.

Evaluation Methodology

System setup

- Gem5 full-system simulator; x86-64 (TSO).
- Ruby memory simulator with Garnet interconnect model.
- 8 out-of-order cores.
- Private L1s; NUCA organization for L2.

Coherence Protocols

- MESI: Standard Ruby implementation.
- TSO-CC: Lazy coherence protocol for TSO [Elver et al., 2014].

Test Generators

McVerSi

- Test size: 1000 operations across all threads.
- Iterations: 10 test executions per test-run.
- GP params: population size 100; tournament selection (size 2); mutation probability 0.005; crossover probability 1.0.

Random

- Test size: 1000 ops
- Iterations: 10

Litmus tests (*diy-litmus*)

- Diy [Alglave et al., 2012] generated litmus tests for TSO (38 tests).

Results

Bugs found within	1 day	5 days	10 days
McVerSi (large mem.)	**100%**	N/A	N/A
Random (small mem.)	73%	73%	73%
Random (large mem.)	55%	73%	82%
diy-litmus	18%	45%	45%

Bugs found, when running up to the equivalent of 10 days time.

Results

Bug	*McVerSi (small mem.)*	*McVerSi (large mem.)*	*Random (small mem.)*	*Random (large mem.)*	*diy-litmus*
MESI,LQ+IS,Inv[1]	**10 (0.01)**	**10 (0.49)**	**10 (0.01)**	**10 (0.89)**	NF
MESI,LQ+SM,Inv[1]	**10 (0.33)**	**10 (5.20)**	**10 (0.48)**	NF	NF
MESI,LQ+E,Inv	**10 (2.97)**	**10 (0.09)**	**10 (4.34)**	**10 (0.10)**	NF
MESI,LQ+M,Inv	**10 (1.42)**	**10 (1.37)**	**10 (1.93)**	**10 (11.05)**	NF
MESI,LQ+S,Replace	NF	**10 (2.69)**	NF	6 (10.10)	NF
MESI+PUTX-Race[2]	NF	**10 (4.64)**	NF	3 (9.63)	NF
MESI+Replace-Race	NF	**10 (0.12)**	NF	**10 (0.19)**	5 (0.53)
TSO-CC+no-epoch-ids	**10 (0.90)**	**10 (7.40)**	**10 (0.96)**	NF	6 (5.93)
TSO-CC+compare	**10 (0.01)**	**10 (2.28)**	**10 (0.01)**	1 (22.31)	**10 (0.92)**
LQ+no-TSO[3]	**10 (0.00)**	**10 (0.03)**	**10 (0.00)**	**10 (0.08)**	**10 (5.35)**
SQ+no-FIFO	**10 (0.01)**	**10 (0.24)**	**10 (0.01)**	**10 (0.40)**	9 (4.77)
All	80 (0.71)	**110 (2.23)**	80 (0.97)	70 (3.41)	40 (3.60)

Bug coverage: *bug found count out of 10 samples (arith. mean hours)*.

[1] Found with McVerSi (patches submitted).

[2] Komuravelli et al., “Revisiting the Complexity of Hardware Cache Coherence and Some Implications”, In: TACO, 2014.

[3] Found in 2013 via litmus testing, patch sent in March 2014; also independently discovered by [Lustig et al., 2014].

Results

Bug	*McVerSi (small mem.)*	*McVerSi (large mem.)*	*Random (small mem.)*	*Random (large mem.)*	*diy-litmus*
MESI,LQ+IS,Inv[1]	**10 (0.01)**	**10 (0.49)**	**10 (0.01)**	**10 (0.89)**	NF
MESI,LQ+SM,Inv[1]	**10 (0.33)**	**10 (5.20)**	**10 (0.48)**	NF	NF
MESI,LQ+E,Inv	**10 (2.97)**	**10 (0.09)**	**10 (4.34)**	**10 (0.10)**	NF
MESI,LQ+M,Inv	**10 (1.42)**	**10 (1.37)**	**10 (1.93)**	**10 (11.05)**	NF
MESI,LQ+S,Replace	NF	**10 (2.69)**	NF	6 (10.10)	NF
MESI+PUTX-Race[2]	NF	**10 (4.64)**	NF	3 (9.63)	NF
MESI+Replace-Race	NF	**10 (0.12)**	NF	**10 (0.19)**	5 (0.53)
TSO-CC+no-epoch-ids	**10 (0.90)**	**10 (7.40)**	**10 (0.96)**	NF	6 (5.93)
TSO-CC+compare	**10 (0.01)**	**10 (2.28)**	**10 (0.01)**	1 (22.31)	**10 (0.92)**
LQ+no-TSO[3]	**10 (0.00)**	**10 (0.03)**	**10 (0.00)**	**10 (0.08)**	**10 (5.35)**
SQ+no-FIFO	**10 (0.01)**	**10 (0.24)**	**10 (0.01)**	**10 (0.40)**	9 (4.77)
All	80 (0.71)	**110 (2.23)**	80 (0.97)	70 (3.41)	40 (3.60)

Bug coverage: *bug found count out of 10 samples (arith. mean hours)*.

[1] Found with McVerSi (patches submitted).

[2] Komuravelli et al., “Revisiting the Complexity of Hardware Cache Coherence and Some Implications”, In: TACO, 2014.

[3] Found in 2013 via litmus testing, patch sent in March 2014; also independently discovered by [Lustig et al., 2014].

Results

Bug	*McVerSi (small mem.)*	*McVerSi (large mem.)*	*Random (small mem.)*	*Random (large mem.)*	*diy-litmus*
MESI,LQ+IS,Inv[1]	**10 (0.01)**	**10 (0.49)**	**10 (0.01)**	**10 (0.89)**	NF
MESI,LQ+SM,Inv[1]	**10 (0.33)**	**10 (5.20)**	**10 (0.48)**	NF	NF
MESI,LQ+E,Inv	**10 (2.97)**	**10 (0.09)**	**10 (4.34)**	**10 (0.10)**	NF
MESI,LQ+M,Inv	**10 (1.42)**	**10 (1.37)**	**10 (1.93)**	**10 (11.05)**	NF
MESI,LQ+S,Replace	NF	**10 (2.69)**	NF	6 (10.10)	NF
MESI+PUTX-Race[2]	NF	**10 (4.64)**	NF	3 (9.63)	NF
MESI+Replace-Race	NF	**10 (0.12)**	NF	**10 (0.19)**	5 (0.53)
TSO-CC+no-epoch-ids	**10 (0.90)**	**10 (7.40)**	**10 (0.96)**	NF	6 (5.93)
TSO-CC+compare	**10 (0.01)**	**10 (2.28)**	**10 (0.01)**	1 (22.31)	**10 (0.92)**
LQ+no-TSO[3]	**10 (0.00)**	**10 (0.03)**	**10 (0.00)**	**10 (0.08)**	**10 (5.35)**
SQ+no-FIFO	**10 (0.01)**	**10 (0.24)**	**10 (0.01)**	**10 (0.40)**	9 (4.77)
All	80 (0.71)	**110 (2.23)**	80 (0.97)	70 (3.41)	40 (3.60)

Bug coverage: *bug found count out of 10 samples (arith. mean hours)*.

[1] Found with McVerSi (patches submitted).

[2] Komuravelli et al., “Revisiting the Complexity of Hardware Cache Coherence and Some Implications”, In: TACO, 2014.

[3] Found in 2013 via litmus testing, patch sent in March 2014; also independently discovered by [Lustig et al., 2014].

Results

Bug	*McVerSi (small mem.)*	*McVerSi (large mem.)*	*Random (small mem.)*	*Random (large mem.)*	*diy-litmus*
MESI,LQ+IS,Inv[1]	**10 (0.01)**	**10 (0.49)**	**10 (0.01)**	**10 (0.89)**	NF
MESI,LQ+SM,Inv[1]	**10 (0.33)**	**10 (5.20)**	**10 (0.48)**	NF	NF
MESI,LQ+E,Inv	**10 (2.97)**	**10 (0.09)**	**10 (4.34)**	**10 (0.10)**	NF
MESI,LQ+M,Inv	**10 (1.42)**	**10 (1.37)**	**10 (1.93)**	**10 (11.05)**	NF
MESI,LQ+S,Replace	NF	**10 (2.69)**	NF	6 (10.10)	NF
MESI+PUTX-Race[2]	NF	**10 (4.64)**	NF	3 (9.63)	NF
MESI+Replace-Race	NF	**10 (0.12)**	NF	**10 (0.19)**	5 (0.53)
TSO-CC+no-epoch-ids	**10 (0.90)**	**10 (7.40)**	**10 (0.96)**	NF	6 (5.93)
TSO-CC+compare	**10 (0.01)**	**10 (2.28)**	**10 (0.01)**	1 (22.31)	**10 (0.92)**
LQ+no-TSO[3]	**10 (0.00)**	**10 (0.03)**	**10 (0.00)**	**10 (0.08)**	**10 (5.35)**
SQ+no-FIFO	**10 (0.01)**	**10 (0.24)**	**10 (0.01)**	**10 (0.40)**	9 (4.77)
All	80 (0.71)	**110 (2.23)**	80 (0.97)	70 (3.41)	40 (3.60)

Bug coverage: *bug found count out of 10 samples (arith. mean hours)*.

[1] Found with McVerSi (patches submitted).

[2] Komuravelli et al., “Revisiting the Complexity of Hardware Cache Coherence and Some Implications”, In: TACO, 2014.

[3] Found in 2013 via litmus testing, patch sent in March 2014; also independently discovered by [Lustig et al., 2014].

Conclusion

Problem

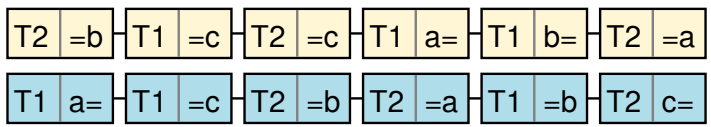
- How to rigorously verify memory consistency model in full system functional design in simulation?

McVerSi: Evolutionary Approach (GP)

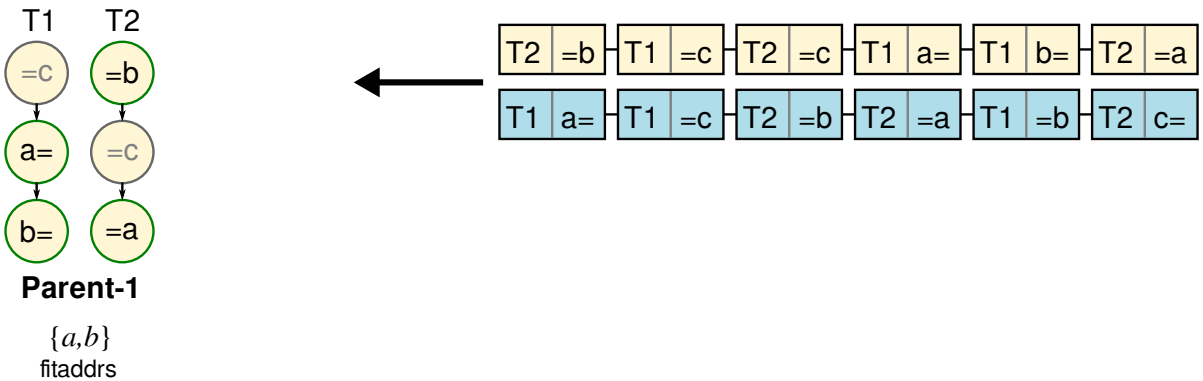
- Fitness: coverage (primary objective).
- Crossover: favoring operations from tests that contribute highly to non-determinism/races (more likely to expose MCM bugs).
- Special guest-host interface to improve throughput (speed up convergence).

- Found 2 new bugs in Gem5 (patches sent upstream).
- Standalone library available: https://github.com/melver/mc2lib

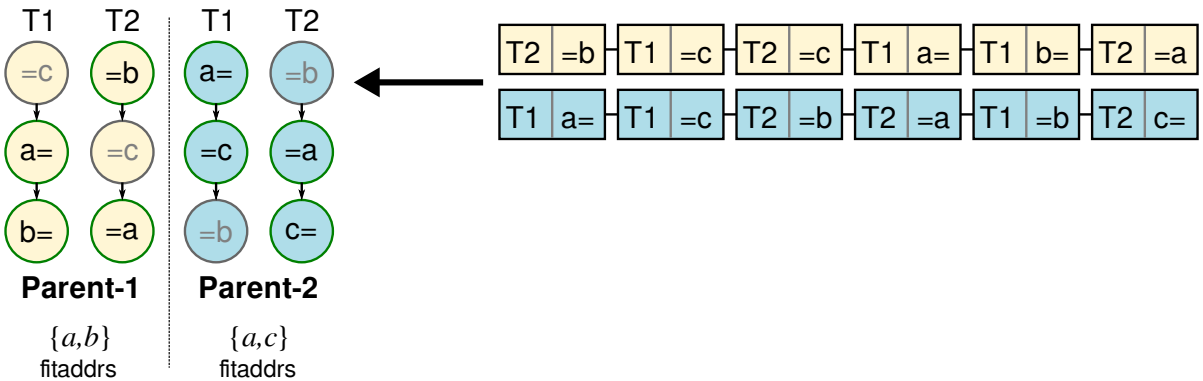
Crossover Example



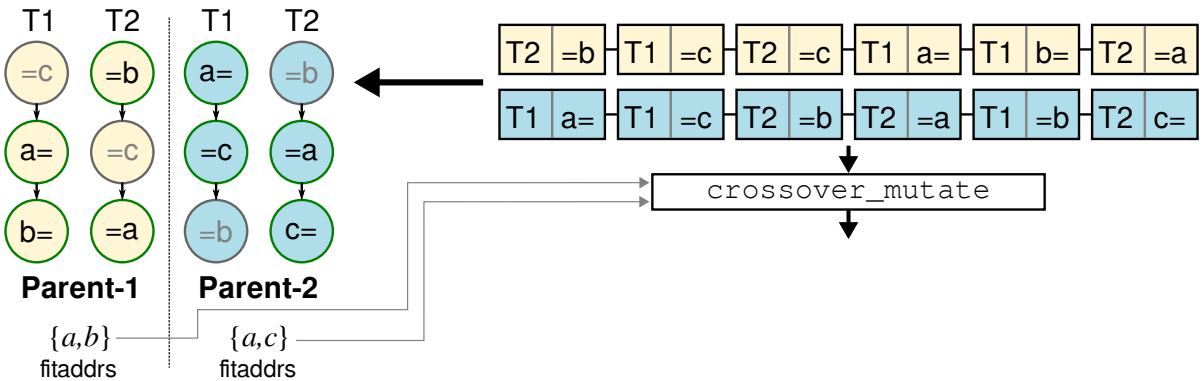
Crossover Example



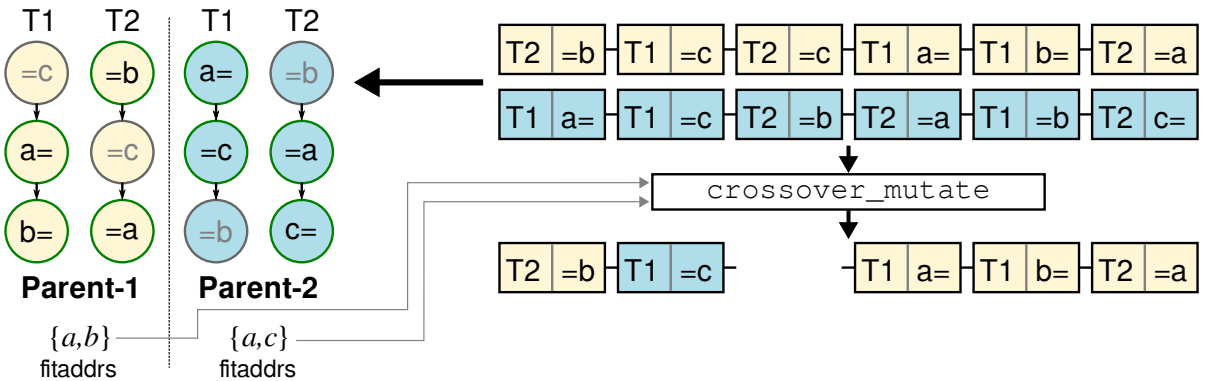
Crossover Example



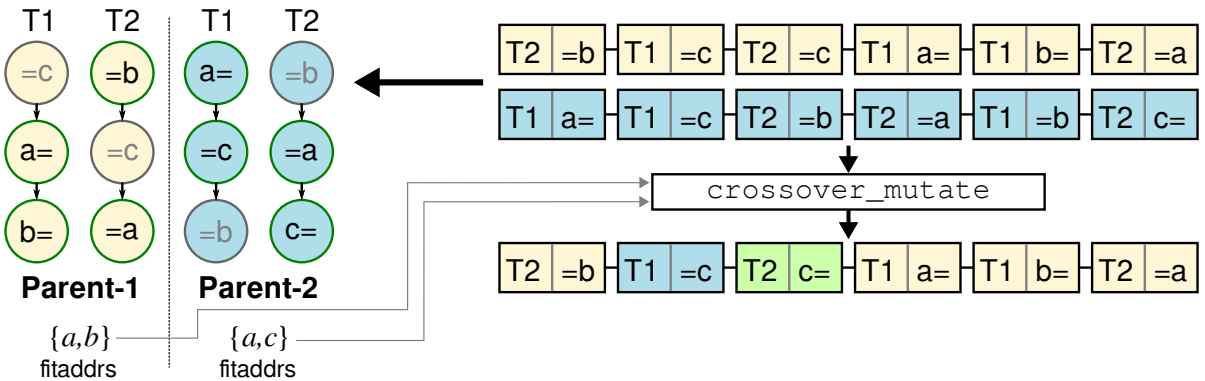
Crossover Example



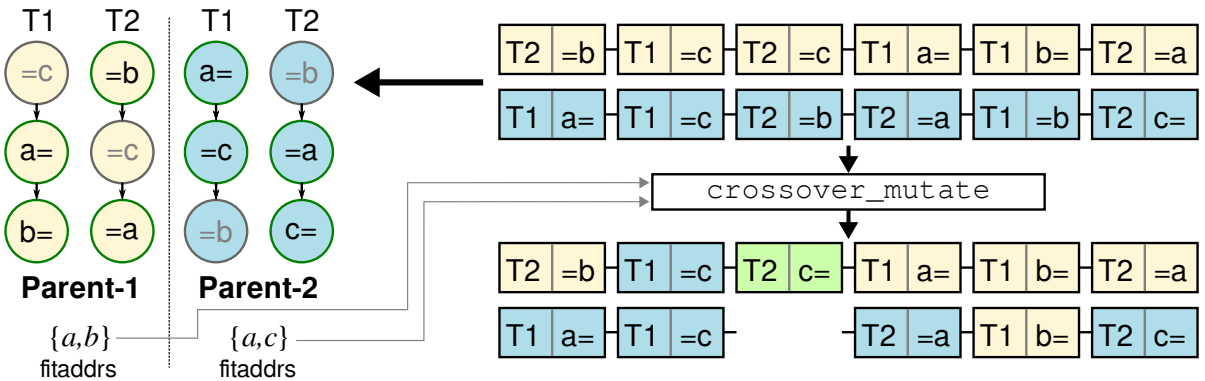
Crossover Example



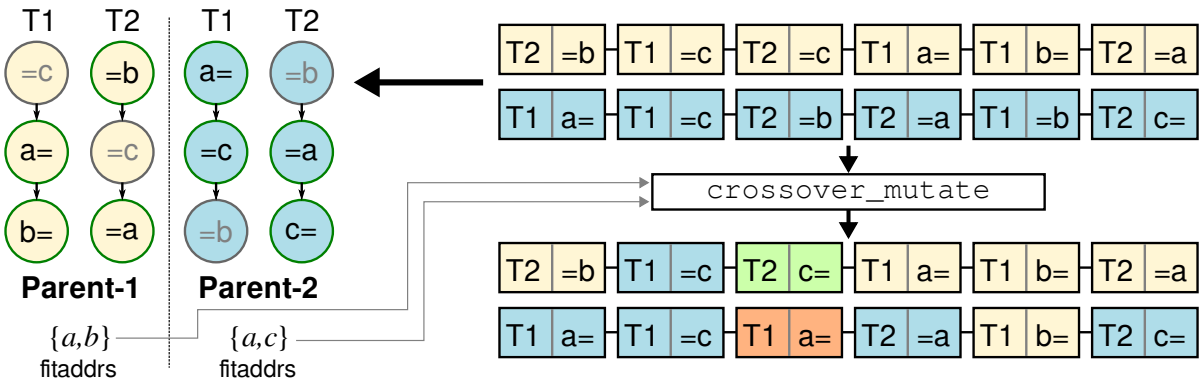
Crossover Example



Crossover Example



Crossover Example



Crossover Example