Concurrent **Bin**ary **E**valuator

Bounded Model Checking of Lockless Programs

Florian Schrögendorfer

florian.schroegendorfer@phlo.at

Introduction

Problem

- heisenbugs due to architecture specific memory ordering habits
- not easily testable and impossible to debug
- especially troublesome for lockless data structures
- requires knowledge about the underlying hardware's characteristics

Goal

Proofing correctness of concurrent programs operating on shared memory.

Approach

- define a simple machine model as an idealized environment
- encode execution of arbitrary programs into SMT formulæ
- proof properties by the means of bounded model checking

Memory Ordering

Memory Ordering

Order of accesses to memory by a CPU at runtime.

Why reorder memory operations? Performance!

- CPUs became so fast that caches simply can't keep up
- unnoticeable by sequential (single threaded) programs
- potentially problematic for parallel programs

	Alpha	ARM	Itanium	MIPS	POWER	x86	zSystems
Loads Reordered after Loads/Stores?	✓	✓	✓	✓	✓		
Stores Reordered after Stores?	✓	✓	✓	✓	✓		
Stores Reordered after Loads?	✓	✓	✓	✓	✓	✓	✓
Atomic Reordered with Loads/Stores?	✓	✓		✓	✓		

Memory Ordering Models

Memory Ordering Model

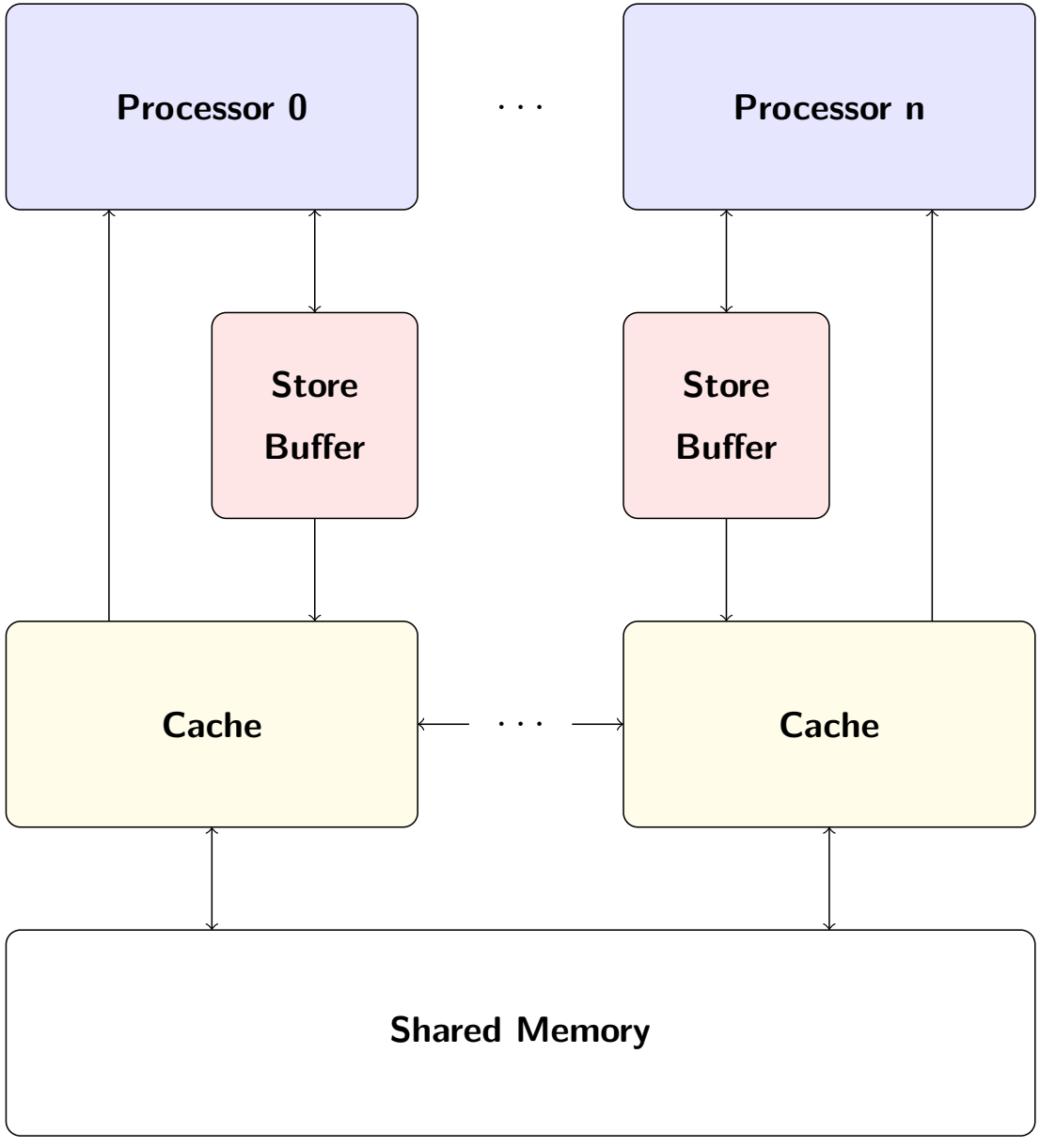
Behaviour of multiprocessor systems regarding memory operations.

- contract between programmer and system
- none of the current major architectures is *sequentially consistent*
- imposes the requirement for explicit memory barrier instructions
- often specified in an informal prose together with litmus tests

Sequential Consistency

A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. [Lamport '79]

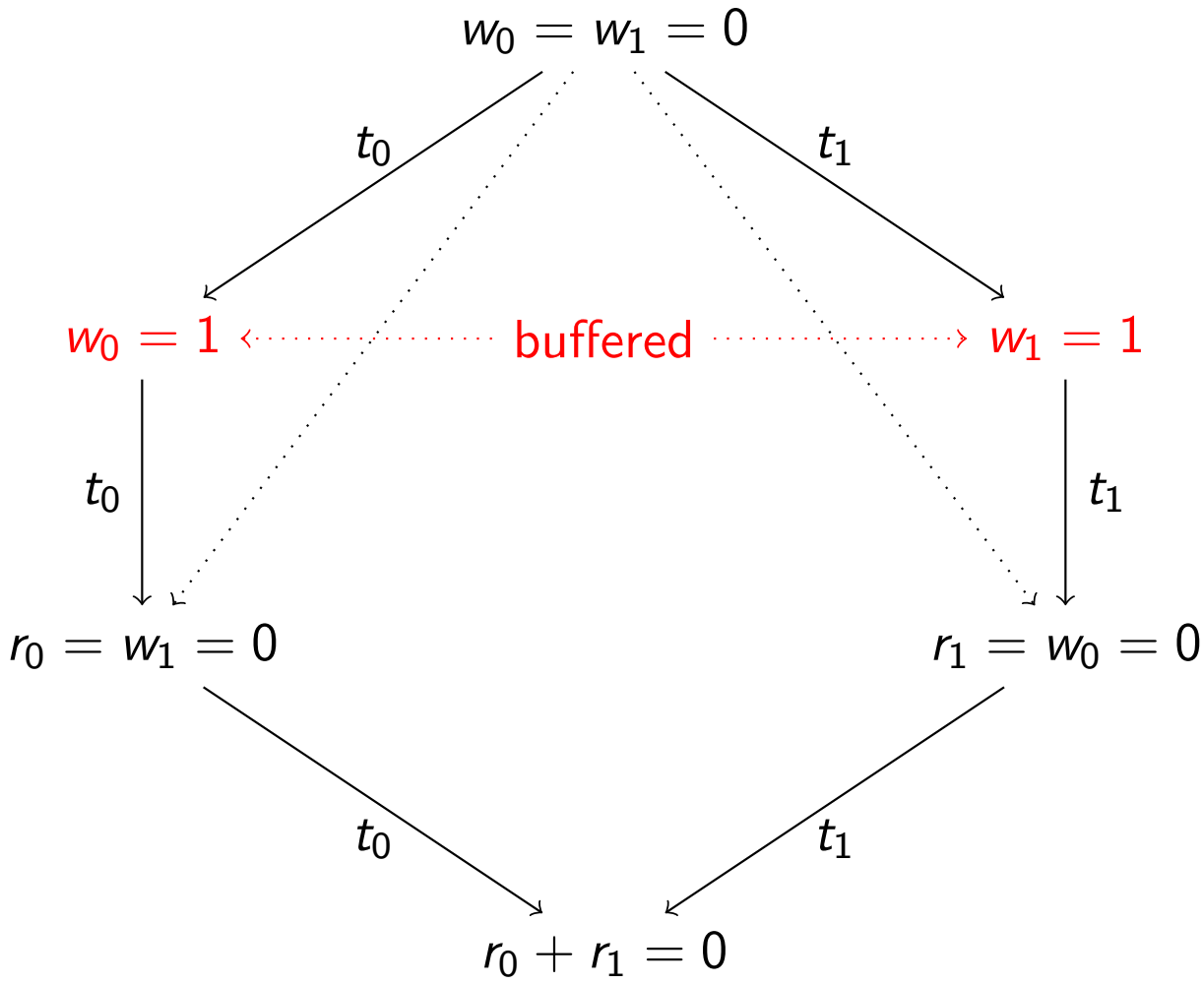
Memory Reordering – Store Buffers



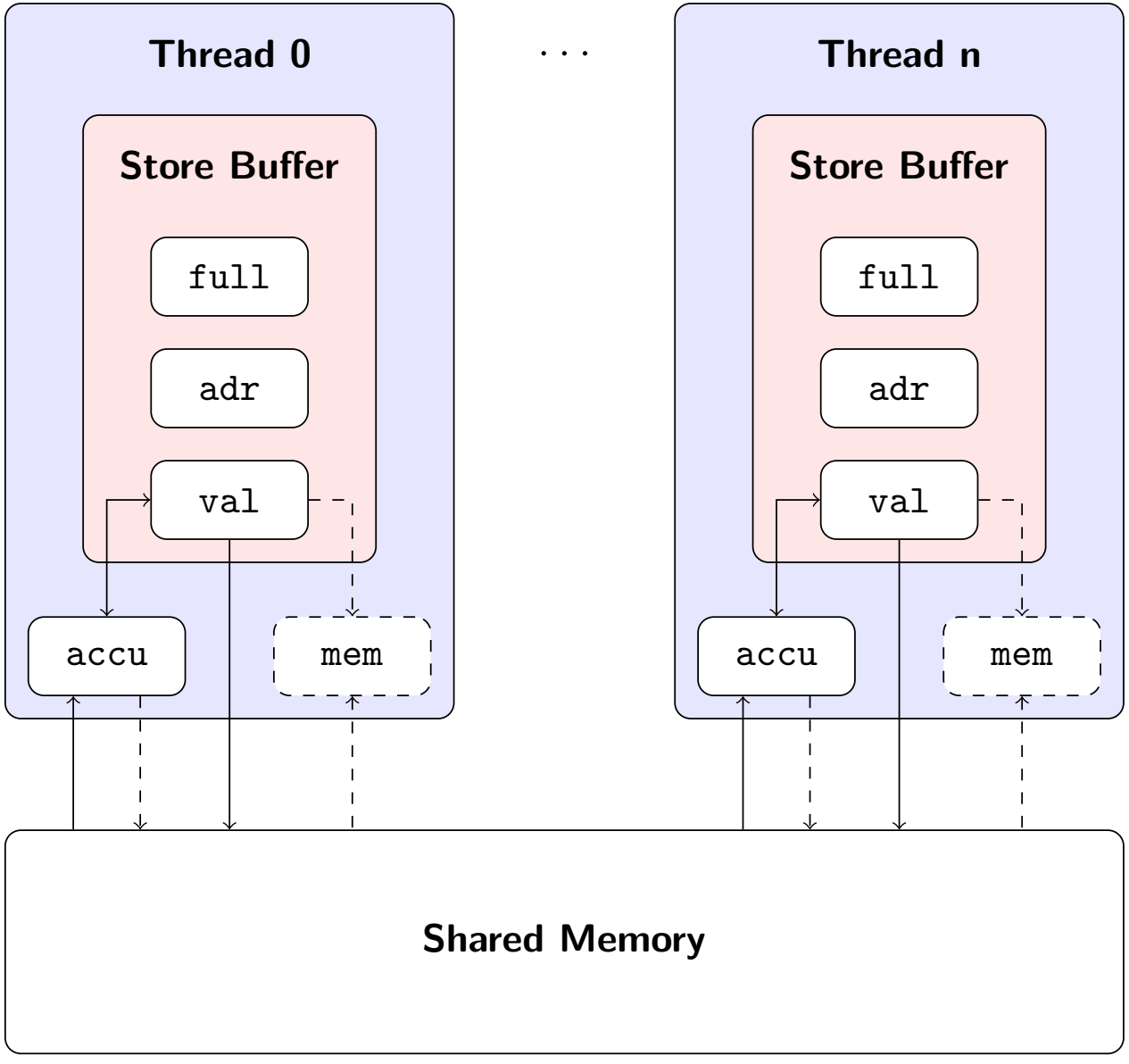
Memory Reordering – Store Buffer Litmus Test

```
1  static int w0 = 0, w1 = 0;
2  static int r0 = 0, r1 = 0;

4  static void * T0 (void * t) {
5    w0 = 1;
6    r0 = w1;
7    return t;
8  }

10 static void * T1 (void * t) {
11   w1 = 1;
12   r1 = w0;
13   return t;
14 }

16 int main () {
17   pthread_t t[2];
18   pthread_create(t + 0, 0, T0, 0);
19   pthread_create(t + 1, 0, T1, 0);
20   pthread_join(t[0], 0);
21   pthread_join(t[1], 0);
22   assert(r0 + r1);
23   return 0;
24 }
```

Memory Reordering – Store Buffer Litmus Test Trace



Virtual Machine Model – Architecture



Virtual Machine Model – Instruction Set

Memory

```
LOAD    adr
STORE   adr
FENCE
```

Atomic

```
MEM     adr
CAS     adr
```

Termination

```
HALT
EXIT    val
```

Arithmetic

```
ADD     adr
ADDI    val
SUB     adr
SUBI    val
MUL     adr
MULI    val
```

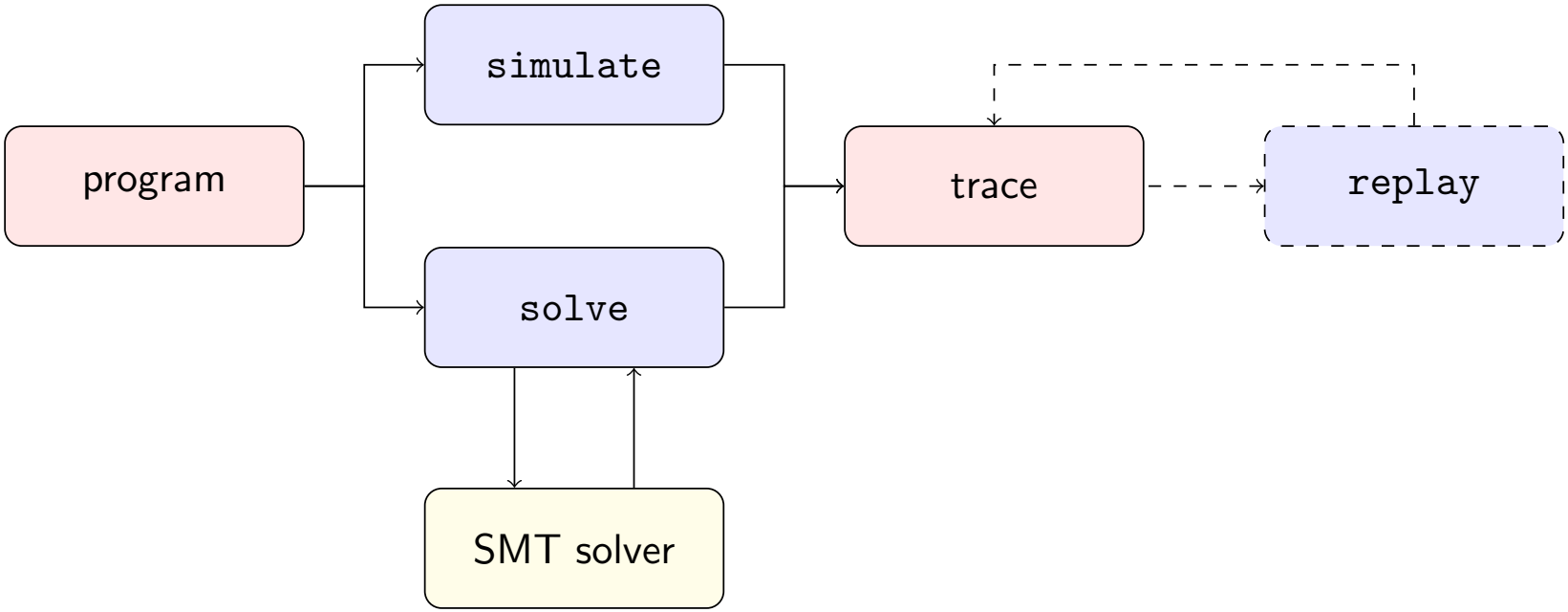
Flow Control

```
CMP     adr
JZ      pc
JNZ     pc
JS      pc
JNS     pc
JNZNS   pc
```

Meta

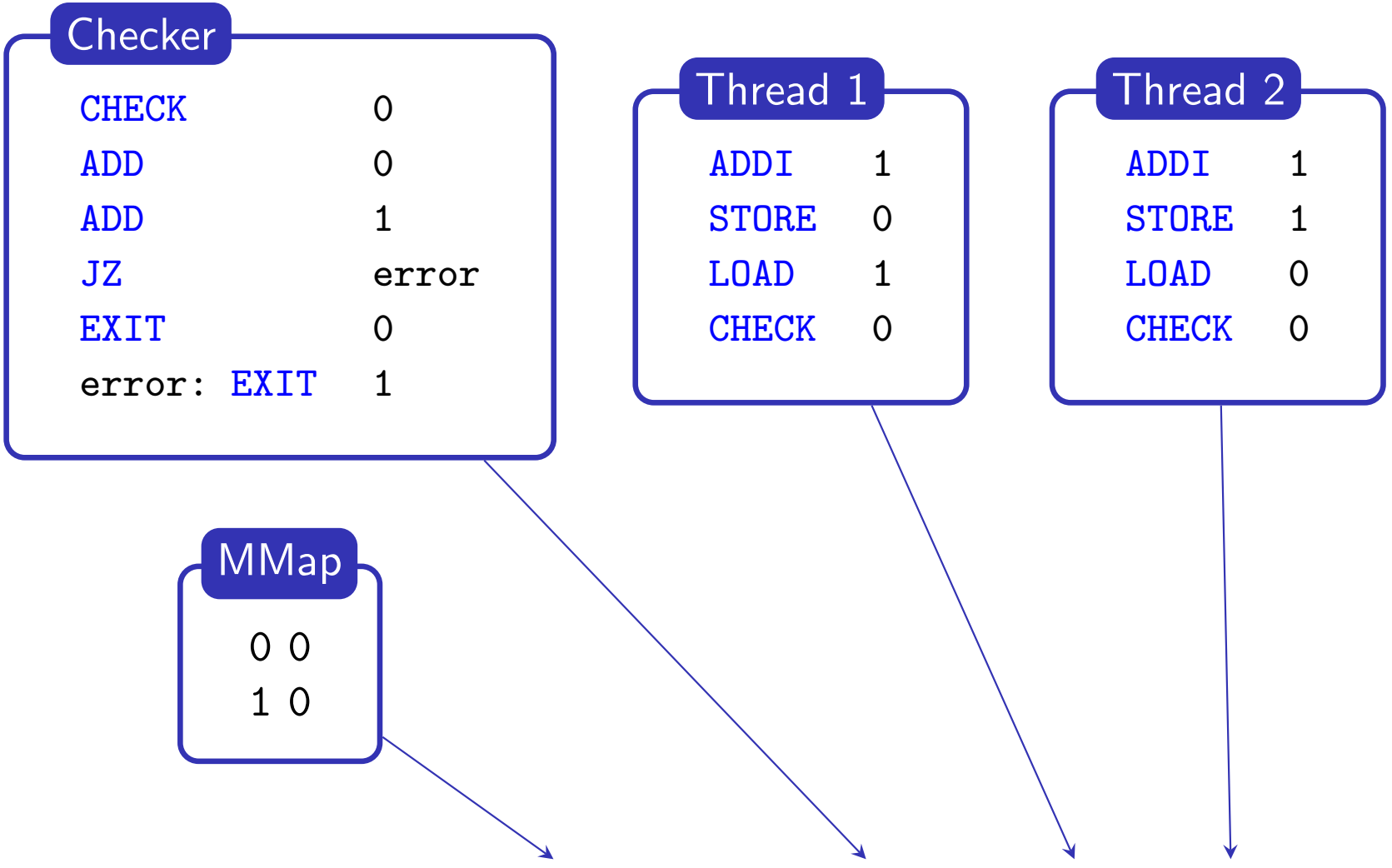
```
CHECK   id
```

ConcuBinE



- `simulate` – simulate execution
- `solve` – encode and solve bounded model checking problem
  - generates SMT-LIB or Btor2 encodings
  - checks for an exit code greater than zero by default
  - uses Boolector (BtorMC), Z3 and CVC4 as backend solvers
  - returs erroneous trace if problem was indeed satisfiable
- `replay` – reevaluate trace via simulation

ConcuBinE – Store Buffer Litmus Test




```
$ concubine solve -m init.mmap 15 checker.asm t1.asm t2.asm
```

ConcuBinE – Store Buffer Litmus Test Trace

```
checker.asm
t1.asm
t2.asm
. smt.mmap
# tid  pc     cmd    arg    accu  mem  adr  val  full  heap
1      0      ADDI   1      0     0    0    0    0     {}   # 0
2      0      ADDI   1      0     0    0    0    0     {}   # 1
1      1      STORE  0      1     0    0    0    0     {}   # 2
2      1      STORE  1      1     0    0    0    0     {}   # 3
1      2      LOAD   1      1     0    0    1    1     {}   # 4
2      2      LOAD   0      1     0    1    1    1     {}   # 5
1      3      CHECK  0      0     0    0    1    1     {}   # 6
2      3      CHECK  0      0     0    1    1    1     {}   # 7
0      0      CHECK  0      0     0    0    0    0     {}   # 8
0      1      ADD    0      0     0    0    0    0     {}   # 9
0      2      ADD    1      0     0    0    0    0     {}   # 10
0      3      JZ     error  0     0    0    0    0     {}   # 11
0      error  EXIT   1      0     0    0    0    0     {}   # 12
```