

Submitted by
Florian Schrögenderfer

Submitted at
**Institute for Formal
Models and Verification**

Supervisor
Armin Biere

August 2021

Bounded Model Checking of Lockless Programs



Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

This printed thesis is identical with the electronic version submitted.

Linz, 24.8.2021

Contents

1	Introduction	1
2	Machine Model	6
3	Simulation	12
4	Encoding Scheme	18
5	SMT-LIB Encoding	26
6	Functional Next State Logic	48
7	Relational Next State Logic	57
8	BTOR2	69
9	Testing and Debugging	81
10	Experiments	82
11	Conclusion	94
	Appendices	96
A	Intel Litmus Tests	96
B	AMD Litmus Tests	106

List of Figures

1	System Architecture with Store Buffers	2
2	Store Buffer Litmus Test Interleavings	4
3	ConcuBinE's Various Modes of Operation	5
4	Abstract Machine Model	6
5	Program Syntax	11
6	Trace Syntax	14
7	Memory Map Syntax	15
8	Intel Litmus Test Encoder Graph	83
9	AMD Litmus Test Encoder Graph	84
10	Intel Litmus Test Solver Graph	85
11	AMD Litmus Test Solver Graph	86
12	Faulty Counter Encoder Graph	89
13	Faulty Counter Solver Graph	90
14	Valid Counter Encoder Graph	92
15	Valid Counter Solver Graph	93

List of Tables

1	Memory Ordering on Different Architectures	1
2	State Variables	18
3	Transition Variables	19
4	Helper Variables	19
5	Frame Axioms	25
6	Store Buffer Litmus Test Programs and Activation Variables .	27
7	SMT-LIB Expression Generator Functions	31
8	Intel Litmus Test Encoder Statistics	82
9	AMD Litmus Test Encoder Statistics	83
10	Intel Litmus Test Solver Statistics	85
11	AMD Litmus Test Solver Statistics	86
12	Faulty Counter Encoder Statistics	88
13	Faulty Counter Solver Statistics	89
14	Valid Counter Encoder Statistics	92
15	Valid Counter Solver Statistics	93
A.1	Stores Are Not Reordered with Other Stores	96
A.2	Stores Are Not Reordered with Older Loads	97
A.3	Loads May be Reordered with Older Stores	98
A.4	Loads Are not Reordered with Older Stores to the Same Location	99
A.5	Intra-Processor Forwarding is Allowed	100
A.6	Stores Are Transitively Visible	101
A.7	Stores Are Seen in a Consistent Order by Other Threads . . .	102
A.8	Locked Instructions Have a Total Order	103
A.9	Loads Are not Reordered with Locks	104
A.10	Stores Are not Reordered with Locks	105
B.1	Stores Are Not Reordered with Other Stores	106
B.2	Stores Are Not Reordered with Older Loads	107
B.3	Stores Can Be Arbitrarily Delayed	107
B.4	Loads May be Reordered with Older Stores	108
B.5	Sequential Consistency	108
B.6	Stores Are Seen in a Consistent Order by Other Threads . . .	109
B.7	Stores Are Transitively Visible	109
B.8	Intra-Processor Forwarding is Allowed	110
B.9	Global Visibility	111

1 Introduction

Correctness of computer systems is critical in today’s information age and although software verification made considerable progress in the last decade, it is still an ongoing research topic. Testing alone however is not sufficient to validate parallel programs communicating via shared memory on modern multiprocessor hardware, as fatal race conditions can have extremely low probabilities of occurrence. The situation becomes even worse due to counter-intuitive behaviour introduced by certain hardware optimizations. Without further knowledge about the underlying architecture, inexperienced developers of concurrent low-level systems code like lockless data structures, operating system kernels, synchronization primitives, compilers, and so on might assume *sequential consistency* [1]. Unfortunately, none of the major hardware architectures follows this rather restrictive *memory ordering model* and allow reordering of memory access operations in various ways. This opens the door for hard to find bugs caused by unexpected behaviour and requires a deep understanding of the target architecture’s memory ordering habits, accompanied by careful use of *memory barrier* operations to ensure consistency.

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

Table 1: Memory Ordering on Different Architectures

Table 1 shows a rough overview of the memory ordering models used by different architectures. Even the most restrictive – like the x86 processors in our desktop computers – allow loads to be reordered with an earlier store to a different location, thus breaking sequential consistency. The reason for this particular reordering is a widely used optimization technique commonly referred to as *store buffer*.

While caches improve subsequent repeated loads of a specific variable (after an initial cache miss), it is also necessary to accommodate frequent concurrent stores from multiple processors to a set of shared variables. In cache-coherent systems, if the caches hold multiple copies of a given variable, all the copies of that variable must have the same value. Each store will therefore invalidate all copies of the old value, hence slowing down the computation [2].

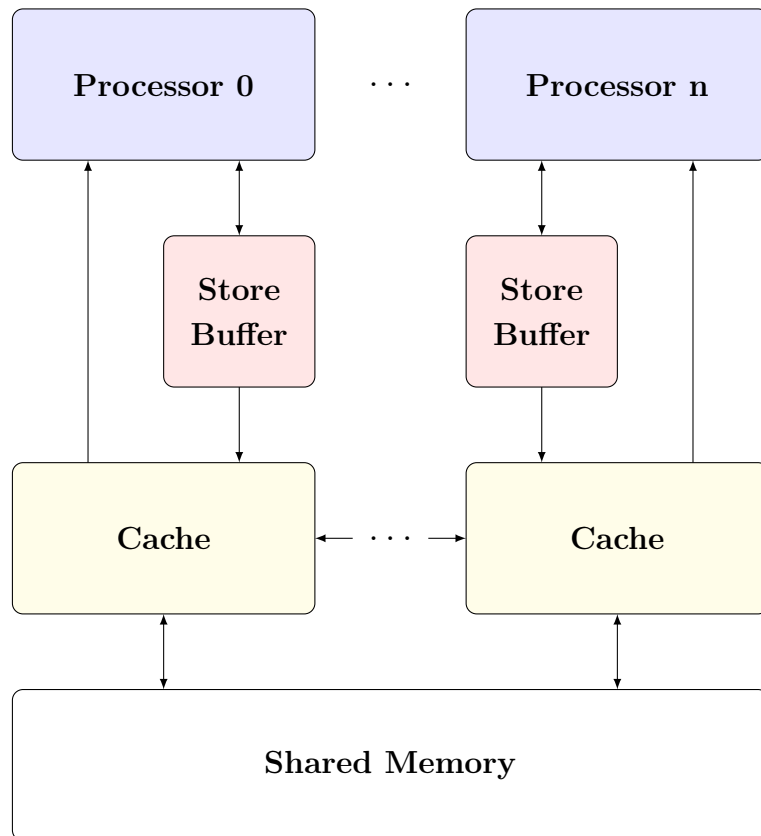


Figure 1: System Architecture with Store Buffers

To remedy this situation, modern processors come equipped with store buffers, as shown in Figure 1. When a variable is stored, the new value is placed in the processor's store buffer, which can proceed immediately without having to wait for the store to do something about all the old values of that variable residing in other processors' caches. In order to guarantee that each processor at least sees its own operations in program order, it first examines its store buffer before consulting the cache when loading a variable in a process called *store forwarding*. Since each store buffer is allowed to voluntarily flush its contents back to memory, the visibility of the particular

stores to other processors might be arbitrarily delayed, leading to the aforementioned memory misordering illustrated by the store buffer litmus test shown in Listing 1.

```
1  #include <assert.h>
2  #include <pthread.h>

4  #define ACCESS(x) (*(volatile typeof(x) *) &(x))
5  #define READ(x) ({typeof(x) TMP = ACCESS(x); TMP;})
6  #define WRITE(x,v) ({ACCESS(x) = (v);})

8  static int w0 = 0;
9  static int w1 = 0;

11 static int r0 = 0;
12 static int r1 = 0;

14 static void * P0 (void * p)
15 {
16     WRITE(w0, 1);
17     r0 = READ(w1);
18     return p;
19 }

21 static void * P1 (void * p)
22 {
23     WRITE(w1, 1);
24     r1 = READ(w0);
25     return p;
26 }

28 int main ()
29 {
30     pthread_t t[2];
31     pthread_create (t + 0, 0, P0, 0);
32     pthread_create (t + 1, 0, P1, 0);
33     pthread_join (t[0], 0);
34     pthread_join (t[1], 0);
35     assert(r0 + r1);
36     return 0;
37 }
```

Listing 1: Store Buffer Litmus Test

One might assume that the assertion on line 35 never triggers, because any possible interleaving should guarantee that at least one `WRITE` must have happened before a subsequent `READ` by the other thread in this symmetric example. This intuition however breaks down with the addition of store buffers. Consider the interleavings depicted in Figure 2. After initializing

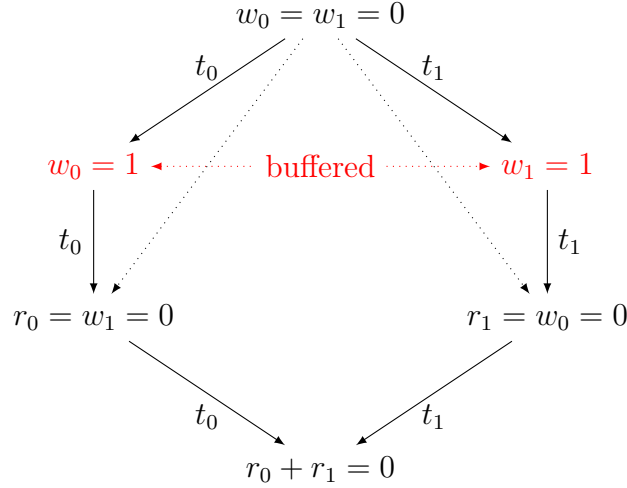


Figure 2: Store Buffer Litmus Test Interleavings

the shared variables with zero, writes to w_0 and w_1 will be placed into the respective processor's store buffer. If neither of the store buffers has been flushed prior to the subsequent reads $r_0 = w_1$ and $r_1 = w_0$, both processors must resort to the variable's initial value in their caches, hence triggering the assertion. This willingness to reorder can be validated by running the store buffer litmus test given in Listing 1. On our Intel i7-3770K this counter-intuitive reordering happened 177169 out of 1000000 test runs, while the perfectly legal outcome of $r_0 + r_1 = 2$ on the other hand only occurred 82 times.

Programmers must therefore make careful use of memory barriers, forcing the store buffer to be flushed back to memory after writing to a shared variable whenever necessary. Unfortunately, many hardware vendors don't supply concrete formal specifications programmers can rely on. For example, the memory ordering models of Intel's [3] as well as AMD's [4] x86 implementations are specified in an informal prose together with a set of litmus tests, having the potential for leading to widespread confusion.

In this thesis we tried to address this issue by means of *bounded model checking* [5] based on the architectural view of a simple virtual machine model using store buffers, reassembling the memory ordering habits of the industry’s leading x86 implementations. In order to evaluate our approach we developed ConcuBinE¹ (short for **C**oncurrent **B**inary **E**valuator), a tool for simulating and automated reasoning about random memory access sequences by concurrent programs. It offers three modes of execution:

- **simulate**: simulates the execution of given programs and returns the corresponding execution trace in an easy to read format.
- **solve**: automatically generates an encoding of given programs for a specific upper bound and evaluates the resulting bounded model checking problem with the help of state-of-the-art SMT solvers. If the problem is satisfiable, the corresponding execution trace (model) will be returned.
- **replay**: reevaluates a given trace via simulation. Used to validate the traces found by the **solve** mode.

Figure 3 shows a basic overview of its usage, where red nodes symbolize in and output files, blue nodes the mode of operation and yellow nodes external software (SMT solvers).

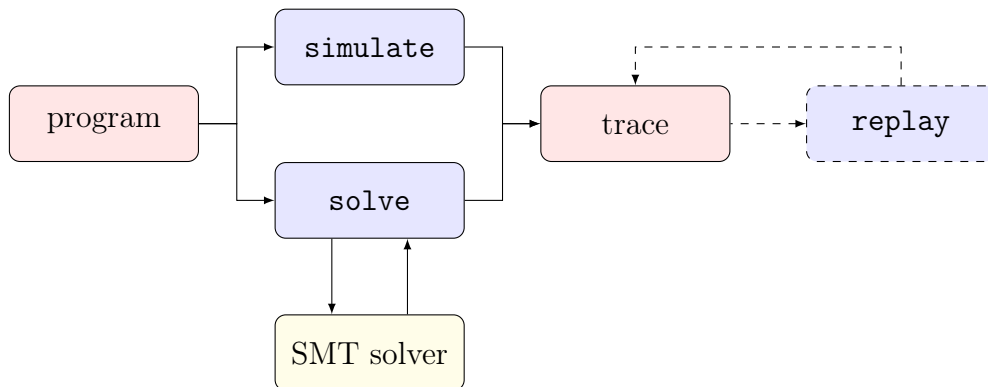


Figure 3: ConcuBinE’s Various Modes of Operation

¹<https://github.com/phlo/concubine>

2 Machine Model

We will start by defining a minimal virtual machine model of a multiprocessor system as observed by assembly programs. To keep the state space of the resulting model checking problems as small as possible, it is based on a 16 bit architecture, using only a minimal set of registers and a radically reduced instruction set.

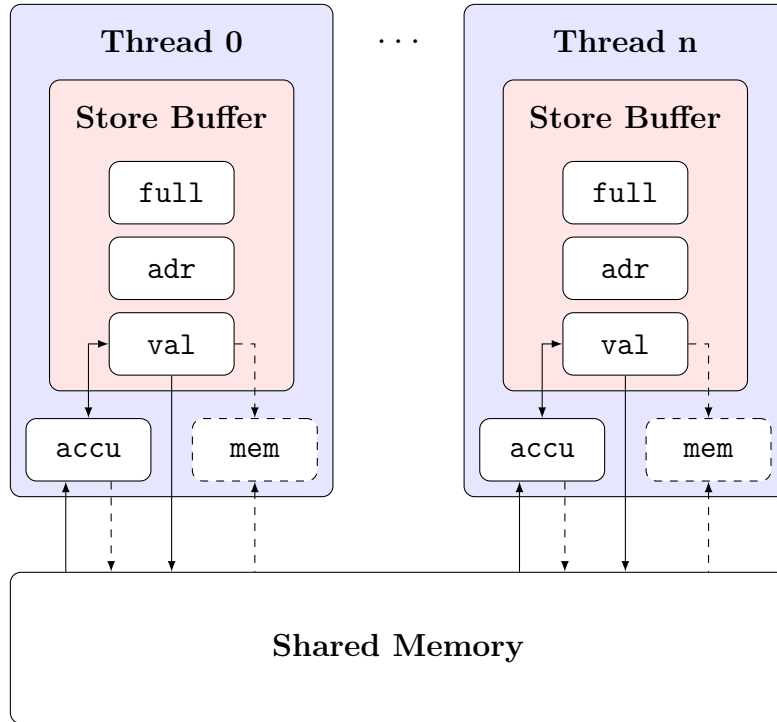


Figure 4: Abstract Machine Model

A schematic overview is illustrated in Figure 4, where dashed lines depict components and data paths solely used by our *compare and swap* mechanism. At the top of the figure are an arbitrary number of logical processors, each running a single abstract thread containing:

- **accu**: a single 16 bit accumulator register
- **mem**: a special purpose 16 bit register, storing the expected value required by a unary *compare and swap* instruction
- a single element *store buffer*, consisting of:

- **full**: a one bit flag register, signaling that it contains a value and may be flushed
- **adr**: a 16 bit address register
- **val**: a 16 bit value register

All threads are directly connected to the machine’s shared memory, referred to as **heap**, which will be uninitialized unless any eventual input data. Caches are abstracted as they don’t influence the memory subsystem’s basic behaviour if they are coherent. In terms of memory ordering, the addition of a *store buffer* allows stores to be reordered after loads, making our model consistent with Intel’s or AMD’s x86 memory ordering models [3, 4].

Scheduling

At each step in time a single thread either executes an instruction or flushes its store buffer back to memory. Scheduling is generally performed non-deterministically under the following constraints.

1. A thread can execute a read, modify or control operation at any time.
2. A thread can voluntarily flush its store buffer to memory only if it is full.
3. A thread can execute a write or barrier operation only if its store buffer is empty.

We argue that this interleaving semantic is sufficient for simulating the effects of real hardware, due to the execution of instructions and store buffer flushes being seen as independent. A similar argument applies regarding preemption, assuming the presence of at least as many physical processors as running threads. Since preemptive operating system kernels contain memory barriers when switching contexts, our thread centric view remains valid and preemption is just hidden behind a store buffer flush.

Instructions

Our machine uses a radically reduced instruction set that contains only the most substantial operations. Instructions are stored separately for each thread and are therefore not contained in memory. This abstraction allows the program counter to address instructions by their index, starting from zero. To simplify the definition of operational semantics, instructions are labelled using the following attributes:

- **modify** – Modifies a register’s content.
- **read** – Reads from memory using *store forwarding*: if **full** is set and **adr** equals the given target address, the value contained in **val** is read instead of the corresponding shared memory location.
- **write** – Writes to the store buffer by setting **full** to true, **adr** to the given target address and **val** to the value contained in **accu**.
- **barrier** – Blocks execution if the store buffer is full (**full** is set).
- **atomic** – Multiple operations performed as a single, indivisible instruction.
- **control** – Modifies the order in which instructions are executed.

Due to the single register architecture, all instructions have at most one operand. The following list shall give an overview of our machine’s instruction set.

Memory

Two addressing modes are supported: direct and indirect denoted by square brackets (e.g. **LOAD** [**arg**]).

LOAD arg	modify, read
------------------------	--------------

Loads the value found at address **arg** into **accu**.

STORE arg	write
-------------------------	-------

Stores the value found in **accu** at address **arg**.

FENCE	barrier
--------------	---------

Memory barrier.

Arithmetic

Since the interpretation of a value’s signedness is left to the programmer, we only included the most basic arithmetic instructions, which are equivalent for unsigned and two’s complement representations. Division is therefore left out due to its different implementations for signed and unsigned integers.

ADD <code>arg</code>	modify, read
-----------------------------	--------------

Adds the value found at address `arg` to `accu`.

ADDI <code>arg</code>	modify
------------------------------	--------

Adds the immediate value `arg` to `accu`.

SUB <code>arg</code>	modify, read
-----------------------------	--------------

Subtracts the value found at address `arg` from `accu`.

SUBI <code>arg</code>	modify
------------------------------	--------

Subtracts the immediate value `arg` from `accu`.

MUL <code>arg</code>	modify, read
-----------------------------	--------------

Multiplies `accu` with the value found at address `arg`.

MULI <code>arg</code>	modify
------------------------------	--------

Multiplies `accu` with the immediate value `arg`.

Control Flow

JMP <code>arg</code>	control
-----------------------------	---------

Jumps to the statement at `arg` unconditionally.

JZ <code>arg</code>	control
----------------------------	---------

Jumps to the statement at `arg` if `accu` is zero.

JNZ <code>arg</code>	control
-----------------------------	---------

Jumps to the statement at `arg` if `accu` is non-zero.

JS <code>arg</code>	control
----------------------------	---------

Jumps to the statement at `arg` if `accu` is negative according to two's complement (most significant bit is set).

JNS `arg` control

Jumps to the statement at `arg` if `accu` is zero or positive according to two's compliment (most significant bit is unset).

JNZNS `arg` control

Jumps to the statement at `arg` if `accu` is positive according to two's compliment (non-zero and most significant bit is unset).

Atomic

We included a basic atomic *compare and swap* operation, commonly used to implement synchronization primitives like semaphores and mutexes, as well as lockless data structures. Since this particular command requires at least two operands (the target address and expected previous value), it had to be split up into two instructions due to our machine's unary input language.

MEM `arg` modify, read

Loads the value from address `arg` into `accu` and `mem` as the expectation during a latter *compare and swap* operation.

CAS `arg` modify, read, barrier, atomic

Atomically compares the expected value in `mem` to the actual value found at address `arg` and only writes the value found in `accu` back to address `adr` if they are equal. Acts like a memory barrier.

Termination

Explicitly stopping a single thread or terminating the whole machine can be achieved by the following commands.

HALT barrier, control

Stops the current thread.

EXIT `arg` control

Stops the machine with exit code `arg`.

Meta

The following high-level meta instruction, mimicking a reverse semaphore (and therefore not available on real hardware), shall simplify the implementation of so called *checker threads* used to programmatically validate machine states at runtime.

CHECK <i>arg</i>	control
-------------------------	---------

Synchronize on checkpoint *arg* (rendezvous). Suspends execution until all threads, containing a call to checkpoint *arg*, reached the corresponding **CHECK** statement.

Programs

Each thread is programmed using an assembly style language, defined by the following syntax.

$\langle int \rangle ::=$ an integer number
 $\langle label \rangle ::=$ a sequence of printable characters without #
 $\langle string \rangle ::=$ a sequence of whitespace and printable characters without # and \n
 $\langle comment \rangle ::=$ # $\langle string \rangle$
 $\langle nullary \rangle ::=$ FENCE | HALT
 $\langle unary \rangle ::=$ ADDI | SUBI | MULI | EXIT | CHECK
 $\langle memory \rangle ::=$ LOAD | STORE | ADD | SUB | MUL | CMP | MEM | CAS
 $\langle jump \rangle ::=$ JMP | JZ | JNZ | JS | JNS | JNZNS
 $\langle instruction \rangle ::=$ $\langle nullary \rangle$
 | $\langle unary \rangle \langle int \rangle$
 | $\langle memory \rangle (\langle int \rangle | [\langle int \rangle])$
 | $\langle jump \rangle (\langle int \rangle | \langle label \rangle)$
 $\langle statement \rangle ::=$ $\langle label \rangle : \langle instruction \rangle$ | $\langle instruction \rangle$
 $\langle line \rangle ::=$ $\langle statement \rangle$ | $\langle statement \rangle \langle comment \rangle$ | $\langle comment \rangle$
 $\langle program \rangle ::=$ $\langle line \rangle$ | $\langle line \rangle \backslash n \langle program \rangle$

Figure 5: Program Syntax

If the final statement in a given program is not an **EXIT** instruction or unconditional **JMP**, an additional **HALT** is inserted implicitly.

3 Simulation

Execution can be simulated using the `simulate` mode, implementing a virtual machine for the previously defined model. It takes an arbitrary number of program files, each being run on a separate thread and produces an execution trace using a pseudo random number generator to determine the schedule.

```
ADDI 1
STORE 0
LOAD 1
CHECK 0
```

Listing 2: `thread.0.asm`

```
ADDI 1
STORE 1
LOAD 0
CHECK 0
```

Listing 3: `thread.1.asm`

An implementation of the store buffer litmus test for our machine is given in Listings 2 and 3. The following command line can now be used to simulate its execution.

```
$ concubine simulate thread.0.asm thread.1.asm
```

After the simulation has been finished, the execution trace will be saved to the current directory as `sim.trace` per default.

Traces

The execution traces are simple text files, capturing the basic environment and machine state transitions. A trace of the previous simulation could look as follows.

```
thread.0.asm
thread.1.asm
.
# tid pc cmd arg accu mem adr val full heap #
0 0 ADDI 1 0 0 0 0 0 0 {} # 0
1 0 ADDI 1 0 0 0 0 0 0 {} # 1
0 1 STORE 0 1 0 0 0 0 0 {} # 2
1 1 STORE 1 1 0 0 0 0 0 {} # 3
0 2 FLUSH - 1 0 0 1 1 {} # 4
1 2 FLUSH - 1 0 1 1 1 {(0,1)} # 5
0 2 LOAD 1 1 0 0 1 0 {(1,1)} # 6
1 2 LOAD 0 1 0 1 1 0 {} # 7
```

0	3	CHECK	0	1	0	0	1	0	{}	# 8
1	3	CHECK	0	1	0	1	1	0	{}	# 9
0	4	HALT	-	1	0	0	1	0	{}	# 10
1	4	HALT	-	1	0	1	1	0	{}	# 11

Listing 4: Simple Output Trace

At the beginning is a list of paths to the programs involved in creating the specific trace, followed by a delimiter. The rest of the file contains the actual execution steps, one per line, starting with the scheduled thread's identifier, followed by details about the state prior to the current statement's execution, consisting of:

- program counter of the instruction about to be executed
- instruction symbol, or **FLUSH** if a thread is flushing its store buffer
- instruction argument, or - for nullary instructions
- accumulator register
- **CAS** memory register
- store buffer address register
- store buffer value register
- store buffer full flag
- memory cell updated in the previous step, represented as a single pair in set notation {(index,value)}, or an empty set {} if the memory state didn't change

The result of the instruction's execution will be visible, the next time the specific thread is scheduled. A better visualization of a particular thread's state transitions can be achieved by only considering e.g. thread 0 in the trace given in Listing 4:

#	tid	pc	cmd	arg	accu	mem	adr	val	full	heap	
0	0		ADDI	1	0	0	0	0	0	{}	# 0
0	1		STORE	0	1	0	0	0	0	{}	# 2
0	2		FLUSH	-	1	0	0	1	1	{}	# 4
0	2		LOAD	1	1	0	0	1	0	{{(1,1)}}	# 6
0	3		CHECK	0	1	0	0	1	0	{}	# 8
0	4		HALT	-	1	0	0	1	0	{}	# 10

A more formal definition of the trace file syntax is given below.

$\langle int \rangle ::=$ an integer number
 $\langle path \rangle ::=$ a unix file path
 $\langle label \rangle ::=$ a sequence of printable characters without #
 $\langle string \rangle ::=$ a sequence of whitespace and printable characters without # and \n
 $\langle comment \rangle ::=$ # $\langle string \rangle$
 $\langle program \rangle ::=$ $\langle path \rangle$ | $\langle path \rangle \langle comment \rangle$ | $\langle comment \rangle$
 $\langle programs \rangle ::=$ $\langle program \rangle$ | $\langle program \rangle \backslash n \langle programs \rangle$
 $\langle mmap \rangle ::=$ $\langle path \rangle$ | $\langle path \rangle \langle comment \rangle$
 $\langle header \rangle ::=$ $\langle programs \rangle \backslash n .$ | $\langle programs \rangle \backslash n . \langle mmap \rangle$
 $\langle pc \rangle ::=$ $\langle int \rangle$ | $\langle label \rangle$
 $\langle op \rangle ::=$ FLUSH
| LOAD | STORE | FENCE
| ADD | ADDI | SUB | SUBI | MUL | MULI
| CMP | JMP | JZ | JNZ | JS | JNS | JNZNS
| MEM | CAS
| HALT | EXIT | CHECK
 $\langle arg \rangle ::=$ $\langle int \rangle$ | [$\langle int \rangle$] | $\langle label \rangle$ | -
 $\langle bool \rangle ::=$ 0 | 1
 $\langle heap \rangle ::=$ {} | {($\langle adr \rangle$, $\langle val \rangle$)}
 $\langle state \rangle ::=$ $\langle int \rangle \langle pc \rangle \langle op \rangle \langle arg \rangle \langle int \rangle \langle int \rangle \langle int \rangle \langle int \rangle \langle bool \rangle \langle heap \rangle$
 $\langle step \rangle ::=$ $\langle state \rangle$ | $\langle state \rangle \langle comment \rangle$ | $\langle comment \rangle$
 $\langle steps \rangle ::=$ $\langle state \rangle$ | $\langle state \rangle \backslash n \langle steps \rangle$
 $\langle trace \rangle ::=$ $\langle header \rangle \backslash n \langle steps \rangle$

Figure 6: Trace Syntax

Memory Maps

Access to uninitialized memory cells is captured in so called *memory maps*² again a simple text file format, containing one address value pair per line, delimited by a whitespace.

```

thread.0.asm
thread.1.asm
. sim.mmap
# tid pc cmd arg accu mem adr val full heap

```

²naming inspired by, but not to be confused with memory mapped I/O

0	0	ADDI	1	0	0	0	0	0	{}	# 0
1	0	ADDI	1	0	0	0	0	0	{}	# 1
0	1	STORE	0	1	0	0	0	0	{}	# 2
0	2	LOAD	1	1	0	0	1	1	{}	# 3
0	3	CHECK	0	57647	0	0	1	1	{}	# 4
1	1	STORE	1	1	0	0	0	0	{}	# 5
1	2	LOAD	0	1	0	1	1	1	{}	# 6
1	3	CHECK	0	34446	0	1	1	1	{}	# 7
0	4	FLUSH	-	57647	0	0	1	1	{}	# 8
1	4	FLUSH	-	34446	0	1	1	1	{{(0,1)}}	# 9
0	4	HALT	-	57647	0	0	1	0	{{(1,1)}}	# 10
1	4	HALT	-	34446	0	1	1	0	{}	# 11

Listing 5: Output Trace Accessing Uninitialized Memory

Consider another random simulation, given in Listing 5. In this simulation, both threads are reading uninitialized memory locations at addresses 0 and 1 respectively. To ensure reproducibility, a memory map file `sim.mmap` is stored together with the trace file `sim.trace` per default and is referenced just after the delimiter at the end of the program list in the trace file's header.

```
0 34446
1 57647
```

Listing 6: Output Memory Map of the Trace in Listing 5

Memory maps can also be used to initialize shared memory with input data by supplying the `-m` command line parameter, followed by a path to an existing memory map file. For completeness, the syntax of such trace files is given below.

```
⟨int⟩ ::= an integer number
⟨string⟩ ::= a sequence of whitespace and printable characters without # and \n
⟨comment⟩ ::= # ⟨string⟩
⟨cell⟩ ::= ⟨int⟩ ⟨int⟩ | ⟨int⟩ ⟨int⟩ ⟨comment⟩ | ⟨comment⟩
⟨mmap⟩ ::= ⟨cell⟩ | ⟨cell⟩ \n ⟨mmap⟩
```

Figure 7: Memory Map Syntax

Finding Specific Machine States

It is also possible to search for specific machine states via simulation. The command line switch `-c` simulates the given programs exhaustively until a trace yielding an exit code other than zero is encountered.

```
CHECK 0
ADD 0
ADD 1
JZ error
EXIT 0
error: EXIT 1
```

Listing 7: `checker.asm`

To find traces exhibiting a problematic reordering of writes in our store buffer litmus test example, we make use of a so called *checker thread*, given in Listing 7, that examines the machine state at a specific point in time by using the `CHECK` instruction and stops execution with an `EXIT 1` if none of the store buffers has been flushed, leading to both memory locations still being zero.

```
0 0
1 0
```

Listing 8: `init.mmap`

This requires that the relevant memory cells are set to zero in order to prevent uninitialized reads from influencing the checking procedure and is achieved by using the memory map given in Listing 8. The search can now be started by issuing the following command line:

```
$ concubine simulate -c -m init.mmap \
    checker.asm \
    thread.0.asm \
    thread.1.asm
```

After a couple of simulations, a trace like the one shown in Listing 9, where none of the store buffers has been flushed prior to the checker thread's `ADD` instructions, will be found.

```

checker.asm
thread.0.asm
thread.1.asm
. sim.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

```

Listing 9: Sequentially Inconsistent Trace

4 Encoding Scheme

The main part of this work is implemented in the `solve` mode, allowing verification of concurrent software running on our abstract machine model by the means of bounded model checking [5]. It takes an arbitrary number of programs plus the upper bound \mathcal{K} as input and encodes them into a finite state machine, expressed as an SMT formula where each transition translates to the execution of a single thread. SMT-LIB [6] and the novel BTOR2 [7] word level model checking format can be generated, using the theories of arrays, uninterpreted functions and bit-vectors. To simplify the definition of bad states directly in the program code, the possibility of encountering an exit code greater than zero is checked for per default, but custom properties may be defined in a separate file and added with the `-c` command line parameter. The resulting formula is then evaluated by a state-of-the-art SMT solver. If it is satisfiable, the corresponding execution trace is extracted from a generated model and stored for later inspection.

Formal Model

Let \mathcal{B}^n be the fixed size bit-vector sort of width n and \mathcal{A}^n the array sort with index and element sorts \mathcal{B}^n . The following variables are used to encode the machine state at a particular step $k \leq \mathcal{K}$, where $k = 0$ is the initial state.

heap^k	$\in \mathcal{A}^{16}$	shared memory
exit^k	$\in \mathcal{B}^1$	exit flag
exit-code^k	$\in \mathcal{B}^{16}$	exit code
accu_t^k	$\in \mathcal{B}^{16}$	accumulator register of thread t
mem_t^k	$\in \mathcal{B}^{16}$	CAS memory register of thread t
adr_t^k	$\in \mathcal{B}^{16}$	store buffer address register of thread t
val_t^k	$\in \mathcal{B}^{16}$	store buffer value register of thread t
full_t^k	$\in \mathcal{B}^1$	store buffer full flag of thread t
$\text{stmt}_{t,pc}^k$	$\in \mathcal{B}^1$	activation flag for statement at pc of thread t
$\text{block}_{id,t}^k$	$\in \mathcal{B}^1$	block flag for checkpoint id of thread t
halt_t^k	$\in \mathcal{B}^1$	halt flag of thread t

Table 2: State Variables

Shared memory states are modelled using the array variables \mathbf{heap}^k in combination with the functions $\mathbf{read}^k: \mathcal{B}^{16} \mapsto \mathcal{B}^{16}$ for retrieving an element and $\mathbf{write}^k: \mathcal{B}^{16} \times \mathcal{B}^{16} \mapsto \mathcal{A}^{16}$ returning an updated version of the shared memory state array with the given element set to a specific value. Register states of a thread t are determined by the bit-vector variables \mathbf{accu}_t^k , \mathbf{mem}_t^k , \mathbf{adr}_t^k , \mathbf{val}_t^k and the flag \mathbf{full}_t^k , signalling that the store buffer is full. To reduce the formula's complexity, program flow is modelled without an explicit program counter ("pc"). Instead, an activation flag $\mathbf{stmt}_{t,pc}^k$ is added for every statement in the program of thread t , expressing that it is about to execute the statement at pc . Blocking a thread t while it is waiting for all other threads to reach a checkpoint id is achieved by a flag \mathbf{block}_t^k . Similarly, the flag \mathbf{halt}_t^k indicates that thread t executed a **HALT** instruction and is therefore also prevented from being scheduled. Termination is captured by the flag \mathbf{exit}^k and bit-vector variable $\mathbf{exit-code}^k$.

All states are initially set to zero, except the initial statement activation flag $\mathbf{stmt}_{t,0}^0$ of every thread t and the shared memory state array \mathbf{heap}^0 , which may be initialized with input data according to a given memory map, but is assumed to be uninitialized in general.

Machine state transitions of the form $s_k \rightarrow^k s_{k+1}$ are encoded by the following free variables.

\mathbf{thread}_t^k	$\in \mathcal{B}^1$	thread t is scheduled to execute an instruction in step k
\mathbf{flush}_t^k	$\in \mathcal{B}^1$	thread t flushes its store buffer in step k

Table 3: Transition Variables

To simplify the definition of successor states, the following helper variables capture frequently used conditions.

$\mathbf{exec}_{t,pc}^k$	$\in \mathcal{B}^1$	thread t is executing the statement at pc in step k
\mathbf{check}_{id}^k	$\in \mathcal{B}^1$	all threads reached checkpoint id in step k

Table 4: Helper Variables

The actual execution of a specific statement is encoded by $\mathbf{exec}_{t,pc}^k$ and is defined as a conjunction of the corresponding statement and thread activation variables.

$$\mathbf{exec}_{t,pc}^k = \mathbf{stmt}_{t,pc}^k \wedge \mathbf{thread}_t^k$$

Furthermore, we use check_{id}^k to signal that all threads containing a call to checkpoint id (given in the set $C_{id} = \{t \mid t \text{ contains } \text{CHECK } id\}$) have synchronized.

$$\text{check}_{id}^k = \bigwedge_{t \in C_{id}} \text{block}_{id,t}^k$$

Scheduling

Non-deterministic scheduling of at most one thread per step is realized by a boolean cardinality constraint over all transition variables and the exit flag exit^k to ensure satisfiability if the machine terminates in a step $k < \mathcal{K}$. Let $\leq_n^1(x_1, \dots, x_n)$ be a predicate expressing that at most one out of n variables is allowed to be true. The intuitive way of encoding $\leq_n^1(x_1, \dots, x_n)$ is by excluding all combinations of two variables being simultaneously true.

$$\bigwedge_{1 \leq i < j \leq n} (\neg x_i \vee \neg x_j)$$

This naïve approach, however, consists of $\binom{n}{2}$ binary clauses. A more compact formulation, based on a sequential counter circuit computing partial sums $s_i = \sum_{j=1}^i x_j$ for increasing values of i up to the final $i = n$, is presented as $\text{LT}_{\text{SEQ}}^{n,1}$ in [8] and defined as follows.

$$(\neg x_1 \vee s_1) \wedge (\neg x_n \vee \neg s_{n-1}) \bigwedge_{1 < i < n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1}))$$

$\text{LT}_{\text{SEQ}}^{n,1}$ is superior to the naïve encoding with regard to the number of clauses for all $n > 5$, as it only requires $3n - 4$ binary clauses and $n - 1$ additional auxiliary variables. The actual definition of the at most one constraint predicate $\leq_n^1(x_1, \dots, x_n)$ is therefore determined by the number of threads involved. Since we have to include two times the number of threads plus one variables in the constraint, the naïve encoding is only used for up to two threads and $\text{LT}_{\text{SEQ}}^{n,1}$ otherwise. An at most one constraint alone is not sufficient, as we need exactly one transition variable or the exit flag to be true in every step. This is because our generated formula should not be trivially satisfiable by never scheduling a single thread. Thus, we define the exactly one constraint predicate $=_n^1(x_1, \dots, x_n)$ by simply adding a disjunction over all variables.

$$(x_1 \vee \dots \vee x_n) \wedge \leq_n^1(x_1, \dots, x_n)$$

If we redeclare n as the number of threads, non-deterministic scheduling of a single thread in step k can now be encoded by the following constraint.

$$=_n^1(\text{thread}_0^k, \dots, \text{thread}_{n-1}^k, \text{flush}_0^k, \dots, \text{flush}_{n-1}^k, \text{exit}^k)$$

This cardinality constraint is further influenced by explicitly disabling transitions from certain states that are prohibited by our machine model. Flushing an empty store buffer of a thread t can be prevented by a simple relational constraint.

$$\text{flush}_t^k \implies \text{full}_t^k$$

In case thread t containing a write, execution of any barrier operation has to be delayed while the store buffer is full. Let F_t be a set of statements requiring an empty store buffer and $\text{ite} : \mathcal{B}^1 \times \mathcal{B}^n \times \mathcal{B}^n \rightarrow \mathcal{B}^n$ be a functional if-then-else, returning $a \in \mathcal{B}^n$ if $x \in \mathcal{B}^1$ is *true*, else $b \in \mathcal{B}^n$.

$$\text{ite}(x, a, b) = \begin{cases} a & \text{if } x \text{ is } \textit{true} \\ b & \text{otherwise} \end{cases}$$

Since both store buffer related constraints mainly depend on mutually exclusive values of full_t^k , we are able to encode them in a single expression.

$$\text{ite}(\text{full}_t^k, (\bigvee_{pc \in F_t} \text{stmt}_{t,pc}^k \implies \neg \text{thread}_t^k), \neg \text{flush}_t^k)$$

Blocking a thread t , while it is waiting for all other threads reaching a checkpoint id is implied by a conjunction of the corresponding block flag $\text{block}_{id,t}^k$ and synchronization variable check_{id}^k .

$$\text{block}_{id,t}^k \wedge \neg \text{check}_{id}^k \implies \neg \text{thread}_t^k$$

Finally, if a thread t has halted, it must also be stopped from being scheduled.

$$\text{halt}_t^k \implies \neg \text{thread}_t^k$$

The purpose of this conditions is to restrict transitions which do not correspond to a valid execution. Since these constraints are defined relationally, special care must be taken in order to prevent a violation of the cardinality constraint, causing the formula to be unsatisfiable if all included variables are falsified simultaneously. Considering the previously defined constraints, the only way this situation might occur is if a *deadlock* is introduced by an unfortunate interleaving of different **CHECK** instructions, resulting in each thread waiting for another. Let T be the set of threads involved and C the set of checkpoint IDs, then the condition causing unsatisfiability can be summarized as follows.

$$\begin{aligned} & \exists k \leq \mathcal{K} : \neg \text{exit}^k \\ & \quad \wedge \\ & \quad \forall t \in T : \neg \text{full}_t^k \wedge \neg \text{halt}_t^k \\ & \quad \wedge \\ & \quad \exists id \in C : \text{block}_{id,t}^k \wedge \neg \text{check}_{id}^k \end{aligned}$$

Memory Access

Due store forwarding, memory access can not be expressed as simple array lookup, but is encoded by a separate function $\text{load}_t^k : \mathcal{B}^{16} \rightarrow \mathcal{B}^{16}$ for loading the shared memory element at address $\text{adr} \in \mathcal{B}^{16}$ with store forwarding from thread t .

$$\begin{aligned} \text{load}_t^k(\text{adr}) = & \text{ite}(\text{full}_t^k \wedge \text{adr}_t^k = \text{adr}, \\ & \text{val}_t^k, \\ & \text{read}^k(\text{adr})) \end{aligned}$$

In case of indirect addressing, load_t^k is redefined to prevent dependency on certain features of the target language or the use of additional auxiliary variables.

$$\text{load}_t^k(\text{adr}) = \text{ite}(\text{full}_t^k, \tag{1}$$

$$\text{ite}(\text{adr}_t^k = \text{adr}, \tag{2}$$

$$\text{ite}(\text{val}_t^k = \text{adr}, \tag{3}$$

$$\text{val}_t^k, \tag{4}$$

$$\text{read}^k(\text{val}_t^k)), \tag{5}$$

$$\text{ite}(\text{adr}_t^k = \text{read}^k(\text{adr}), \tag{6}$$

$$\text{val}_t^k, \tag{7}$$

$$\text{read}^k(\text{read}^k(\text{adr}))), \tag{8}$$

$$\text{read}^k(\text{read}^k(\text{adr}))) \tag{9}$$

First, we check if the store buffer contains an entry and store forwarding might apply (1). If it is empty, the requested value has to be directly retrieved from memory (9). Otherwise, if the store buffer contains an entry for the given address (2), we must further check if it is equal to the effective address (3) to determine if either both (4), or just the given address can be forwarded (5). Finally, in case there is no entry for the given address, the effective address is fetched from memory and compared to the store buffer address (6) to see if it might be contained (7), or nothing can be forwarded after all (8).

Frame Axioms

Successor states of transitions for all possible types of operations, determined by flush_t^k and $\text{exec}_{t,pc}^k$, are defined by the frame axioms in the table below. All state variables, not explicitly altered are assumed to be unchanged in

the next step, except the block flags $\text{block}_{id,t}^{k+1}$, which are reset if all threads synchronized upon checkpoint id .

$$\text{block}_{id,t}^{k+1} = \text{ite}(\text{check}_{id}^k, \text{false}, \text{block}_{id,t}^k)$$

To further simplify the definition of axioms, two additional functions are introduced: $\text{msb} : \mathcal{B}^{16} \rightarrow \mathcal{B}^1$ for retrieving the most significant bit of a given bit-vector and $\text{effective}^k : \mathcal{B}^{16} \rightarrow \mathcal{B}^{16}$ for transparently selecting the effective address during **STORE** or **CAS** instructions.

$$\text{effective}^k(adr) = \begin{cases} \text{read}^k(adr) & \text{if indirect} \\ adr & \text{otherwise} \end{cases}$$

FLUSH	heap^{k+1}	$= \text{write}^k(\text{adr}_t^k, \text{val}_t^k)$
	full^{k+1}	$= \text{false}$
LOAD arg	accu_t^{k+1}	$= \text{load}_t^k(\text{arg})$
	$\text{stmt}_{t,pc}^{k+1}$	$= \text{false}$
	$\text{stmt}_{t,pc+1}^{k+1}$	$= \text{true}$
STORE arg	adr_t^{k+1}	$= \text{effective}^k(\text{arg})$
	val_t^{k+1}	$= \text{accu}_t^k$
	$\text{stmt}_{t,pc}^{k+1}$	$= \text{false}$
	$\text{stmt}_{t,pc+1}^{k+1}$	$= \text{true}$
FENCE	$\text{stmt}_{t,pc}^{k+1}$	$= \text{false}$
	$\text{stmt}_{t,pc+1}^{k+1}$	$= \text{true}$
ADD arg	accu_t^{k+1}	$= \text{accu}_t^k + \text{load}_t^k(\text{arg})$
	$\text{stmt}_{t,pc}^{k+1}$	$= \text{false}$
	$\text{stmt}_{t,pc+1}^{k+1}$	$= \text{true}$
ADDI arg	accu_t^{k+1}	$= \text{accu}_t^k + \text{arg}$
	$\text{stmt}_{t,pc}^{k+1}$	$= \text{false}$
	$\text{stmt}_{t,pc+1}^{k+1}$	$= \text{true}$

SUB arg	accu_t^{k+1}	$= \text{accu}_t^k - \text{load}_t^k(\text{arg})$
	$\text{stmt}_{t,pc}^{k+1}$	$= \text{false}$
	$\text{stmt}_{t,pc+1}^{k+1}$	$= \text{true}$
SUBI arg	accu_t^{k+1}	$= \text{accu}_t^k - \text{arg}$
	$\text{stmt}_{t,pc}^{k+1}$	$= \text{false}$
	$\text{stmt}_{t,pc+1}^{k+1}$	$= \text{true}$
MUL arg	accu_t^{k+1}	$= \text{accu}_t^k * \text{load}_t^k(\text{arg})$
	$\text{stmt}_{t,pc}^{k+1}$	$= \text{false}$
	$\text{stmt}_{t,pc+1}^{k+1}$	$= \text{true}$
MULI arg	accu_t^{k+1}	$= \text{accu}_t^k * \text{arg}$
	$\text{stmt}_{t,pc}^{k+1}$	$= \text{false}$
	$\text{stmt}_{t,pc+1}^{k+1}$	$= \text{true}$
CMP arg	accu_t^{k+1}	$= \text{accu}_t^k - \text{load}_t^k(\text{arg})$
	$\text{stmt}_{t,pc}^{k+1}$	$= \text{false}$
	$\text{stmt}_{t,pc+1}^{k+1}$	$= \text{true}$
JMP arg	$\text{stmt}_{t,pc}^{k+1}$	$= \text{ite}(pc \neq \text{arg}, \text{false}, \text{true})$
	$\text{stmt}_{t,\text{arg}}^{k+1}$	$= \text{ite}(pc \neq \text{arg}, \text{true}, \text{false})$
JZ arg	$\text{stmt}_{t,pc}^{k+1}$	$= \text{ite}(pc \neq \text{arg}, \text{false}, \neg \text{accu}_t^k)$
	$\text{stmt}_{t,pc+1}^{k+1}$	$= \text{accu}_t^k$
	$\text{stmt}_{t,\text{arg}}^{k+1}$	$= \neg \text{accu}_t^k$
JNZ arg	$\text{stmt}_{t,pc}^{k+1}$	$= \text{ite}(pc \neq \text{arg}, \text{false}, \text{accu}_t^k)$
	$\text{stmt}_{t,pc+1}^{k+1}$	$= \neg \text{accu}_t^k$
	$\text{stmt}_{t,\text{arg}}^{k+1}$	$= \text{accu}_t^k$
JS arg	$\text{stmt}_{t,pc}^{k+1}$	$= \text{ite}(pc \neq \text{arg}, \text{false}, \text{msb}(\text{accu}_t^k))$
	$\text{stmt}_{t,pc+1}^{k+1}$	$= \neg \text{msb}(\text{accu}_t^k)$
	$\text{stmt}_{t,\text{arg}}^{k+1}$	$= \text{msb}(\text{accu}_t^k)$

JNS	arg	$\text{stmt}_{t,pc}^{k+1}$	$= \text{ite}(pc \neq arg, false, \neg \text{msb}(\text{accu}_t^k))$
		$\text{stmt}_{t,pc+1}^{k+1}$	$= \text{msb}(\text{accu}_t^k)$
		$\text{stmt}_{t,arg}^{k+1}$	$= \neg \text{msb}(\text{accu}_t^k)$
JNZNS	arg	$\text{stmt}_{t,pc}^{k+1}$	$= \text{ite}(pc \neq arg, false, \text{accu}_t^k \wedge \neg \text{msb}(\text{accu}_t^k))$
		$\text{stmt}_{t,pc+1}^{k+1}$	$= \neg \text{accu}_t^k \vee \text{msb}(\text{accu}_t^k)$
		$\text{stmt}_{t,arg}^{k+1}$	$= \text{accu}_t^k \wedge \neg \text{msb}(\text{accu}_t^k)$
MEM	arg	accu_t^{k+1}	$= \text{load}_t^k(\text{arg})$
		mem_t^{k+1}	$= \text{load}_t^k(\text{arg})$
		$\text{stmt}_{t,pc}^{k+1}$	$= false$
		$\text{stmt}_{t,pc+1}^{k+k}$	$= true$
CAS	arg		$\text{ite}(\text{mem}_t^k = \text{read}^k(\text{effective}^k(\text{arg})),$
		heap^{k+1}	$= \text{write}^k(\text{effective}^k(\text{arg}), \text{accu}_t^k),$
			$\text{heap}^k)$
		accu_t^{k+1}	$= \text{ite}(\text{mem}_t^k = \text{read}^k(\text{effective}^k(\text{arg})), 1, 0)$
		$\text{stmt}_{t,pc}^{k+1}$	$= false$
HALT		$\text{stmt}_{t,pc+1}^{k+1}$	$= true$
		exit_t^{k+1}	$= \bigwedge_{i=0}^{n-1} \text{halt}_i^{k+1}$
		halt_t^{k+1}	$= true$
EXIT	arg	$\text{stmt}_{t,pc}^{k+1}$	$= false$
		exit_t^{k+1}	$= true$
		exit-code^k	$= \text{arg}$
CHECK	arg	$\text{stmt}_{t,pc}^{k+1}$	$= false$
		$\text{stmt}_{t,pc+1}^{k+1}$	$= true$
		$\text{block}_{arg,t}^{k+1}$	$= true$

Table 5: Frame Axioms

5 SMT-LIB Encoding

We will now outline a simplistic C++17 implementation for encoding our bounded model checking problems in the SMT-LIB [6] format according to the previously defined scheme and use the store buffer litmus test as an exemplary input for demonstration. SMT-LIB is the result of an ongoing international initiative with the goal of developing a rigorously standardized language for describing SMT theories as well as formulas and is supported by nearly all SMT solvers. Based on a Lisp like syntax, it covers a wide variety of theories like the theory of integers, fixed size bit-vectors, arrays and many more. In order to apply specialized and more efficient satisfiability techniques, theories are grouped into so called logics and every formula must explicitly specify the logic it is based on. See www.smt-lib.org for a full list of theories and logics available. In our case `QF_AUFBV` was used, allowing closed quantifier-free formulas over the theory of bit-vectors and bit-vector arrays extended with free sort and function symbols.

SMT-LIB also offers incremental solving through `push` and `pop` commands. Because of our encoding's iterative nature, it would have been the best choice in order to keep the size of our formulas as small as possible. Unfortunately, incremental solving just started to be supported by the main SMT solvers at the time we began development and therefore chose to manually unroll our bounded model checking problem for every step $k \leq \mathcal{K}$.

Types

To differentiate among particular machine states, we start by introducing:

```
enum State
{
    heap,
    accu,
    mem,
    adr,
    val,
    full,
    stmt,
    block,
    halt,
    exit,
    exit_code
};
```

Each available instruction is represented by its own type, derived from an abstract class `Instruction`, capturing arguments and abolishing the need for lengthy case splits by utilizing dynamic dispatch of virtual member functions for taking the appropriate action at runtime.

```
struct Instruction
{
    uint arg;
    bool indirect;

    Instruction (uint a, bool i = false) : arg(a), indirect(i) {};

    virtual std::string encode (uint k, uint t, State state) = 0;
};
```

Globals

Commonly used variables are given as globals to keep things simple and function signatures as small as possible.

Thread 0	Thread 1	Thread 2
ADDI 1 stmt _{0,0} ^k	ADDI 1 stmt _{1,0} ^k	
STORE 0 stmt _{0,1} ^k	STORE 1 stmt _{1,1} ^k	
LOAD 1 stmt _{0,2} ^k	LOAD 0 stmt _{1,2} ^k	
CHECK 0 stmt _{0,3} ^k	CHECK 0 stmt _{1,3} ^k	CHECK 0 stmt _{2,0} ^k
HALT stmt _{0,4} ^k	HALT stmt _{1,4} ^k	ADD 0 stmt _{2,1} ^k
		ADD 1 stmt _{2,2} ^k
		JZ error stmt _{2,3} ^k
		EXIT 0 stmt _{2,4} ^k
		error: EXIT 1 stmt _{2,5} ^k

Table 6: Store Buffer Litmus Test Programs and Activation Variables

Input programs are expressed as lists of instruction pointers and initialized according to our store buffer litmus test, given in Table 6.


```

std::vector<std::vector<Instruction *>> programs = {
    {
        new Addi(1),
        new Store(0),
        new Load(1),
        new Check(0),
        new Halt()
    },
    {
        new Addi(1),
        new Store(1),
        new Load(0),
        new Check(0),
        new Halt()
    },
    {
        new Check(0),
        new Add(0),
        new Add(1),
        new Jz(5),
        new Exit(0),
        new Exit(1)
    }
};

```

The initial memory layout in our example, given by the memory map in Listing 8, is captured by the following variable.

```
std::map<uint, uint> mmap = {{0, 0}, {1, 0}};
```

Last but not least, we define the upper bound \mathcal{K} of the resulting model checking problem as the final input variable.

```
uint bound = 17;
```

The resulting formula is stored in a string stream to increase readability by the use of compound statements.

```
std::ostringstream formula;
```

To simplify the encoding process, the following utility variables are generated during a preprocessing step by analyzing the input programs. Although they might seem a bit complex, we omit further details on how they are created as they just identify certain types of instructions and initialize them according to our store buffer litmus test example for completeness.

- Program counters of instructions modifying a particular state.

```
std::map<State, std::vector<std::vector<uint>>> updates = {
    {State::accu, {{0, 2},      // accu0 ← ADDI 1, LOAD 0
                  {0, 2},      // accu1 ← ADDI 1, LOAD 1
                  {1, 2}}},    // accu2 ← ADD 0, ADD 1
    {State::adr,  {{1},        // adr0 ← STORE 0
                  {1},        // adr1 ← STORE 1
                  {}},        // adr2 ← ∅
    {State::val,  {{1},        // val0 ← STORE 0
                  {1},        // val1 ← STORE 1
                  {}},        // val2 ← ∅
    {State::full, {{1},        // full0 ← STORE 0
                  {1},        // full1 ← STORE 1
                  {}},        // full2 ← ∅
    {State::halt, {{4},        // halt0 ← HALT
                  {4},        // halt1 ← HALT
                  {}},        // halt2 ← ∅
    {State::exit, {},         // exit ← ∅
                  {},         // exit ← ∅
                  {4, 5}}}}; // exit ← EXIT 0, EXIT 1
```

- Predecessors of each statement.

```
std::vector<std::vector<std::vector<uint>>> predecessors = {
    // thread 0
    {{},      // ADDI 1    ← ∅
     {0},     // STORE 0   ← ADDI 1
     {1},     // LOAD 1    ← STORE 0
     {2},     // CHECK 0    ← LOAD 1
     {3}},    // HALT      ← CHECK 0
    // thread 1
    {{},      // ADDI 1    ← ∅
     {0},     // STORE 1   ← ADDI 1
     {1},     // LOAD 0    ← STORE 1
     {2},     // CHECK 0    ← LOAD 0
     {3}},    // HALT      ← CHECK 0
    // thread 2
    {{},      // CHECK 0    ← ∅
     {0},     // ADD 0      ← CHECK 0
     {1},     // ADD 1      ← ADD 0
     {2},     // JZ error   ← ADD 1
     {3},     // EXIT 0      ← JZ error
     {3}}}}; // error: EXIT 1 ← JZ error
```

- Program counters of memory barriers.

```
std::vector<std::vector<uint>>> barriers = {
    {1, 4}, // thread 0: STORE 0, HALT
    {1, 4}, // thread 1: STORE 1, HALT
    {}};    // thread 2: ∅
```

- Program counters of `CHECK` instructions per checkpoint and thread.

```
std::map<uint, std::map<uint, std::vector<uint>>>> checkpoints = {
    // checkpoint 0
    {0, {{0, {3}}, // thread 0: CHECK 0
        {1, {3}}, // thread 1: CHECK 0
        {2, {0}}}}, // thread 2: CHECK 0
```

SMT-LIB Generator Functions

Frequently used SMT-LIB expressions are generated by a set of functions, based on the following variadic function template as the basic expression generator.

```
template <class ... T>
std::string expr (const std::string & op, const T & ... args)
{
    std::string e = '(' + op;
    (((e += ' ') += args), ...);
    return e += ')';
}
```

Generator functions for SMT-LIB commands supporting a variable number of arguments by being defined as either `:left-assoc`, `:right-assoc` or `:chainable` also include two additional overloads: a variadic version for an arbitrary but fixed number of arguments, using the same template definition as the basic expression generator `expr` and another for handling a varying number of arguments at runtime, based on the following template accepting an arbitrary STL container. In order to emit syntactically correct SMT-LIB expressions, both handle calls containing only a single argument by directly returning it.

```

template <template <class, class ...> class C>
std::string expr (const std::string & op, const C<std::string> & args)
{
    std::string e = '(' + op;
    for (const auto & a : args)
        (e += ' ') += a;
    return e += ')';
}

```

Table 7 shows all generator function names and the corresponding SMT-LIB command they are creating.

declare_bool declare_bv declare_array	declare boolean declare bit-vector declare array	(declare-fun ...)
assertion	assertion	(assert ...)
lnot land lor lxor imply equal ite	negation conjunction disjunction exclusive disjunction implication equivalence functional if-then-else	(not ...) (and ...) (or ...) (xor ...) (=> ...) (= ...) (ite ...)
bvadd bvsub bvmul	bit-vector addition bit-vector subtraction bit-vector multiplication	(bvadd ...) (bvsub ...) (bvmul ...)
select store extract	array read array store bit-vector extraction	(select ...) (store ...) (extract ...)

Table 7: SMT-LIB Expression Generator Functions

Finally, we introduce a function `std::string consth (uint val)` for generating hexadecimal bit-vector constants and a helper function to simplify variable assignments.

```

std::string assign (std::string var, std::string val)
{
    return assertion(equal(var, val));
}

```

Common Encoding Functions

We will now give a top-down overview of the actual encoding process implemented in `src/encoder_smtlib.cc` and start with the main encoding function defined below.

```
void encode ()
{
    formula << "(set-logic QF_AUFBV)\n";

    for (uint k = 0; k <= bound; k++)
    {
        declare_states(k);
        declare_transitions(k);
        define_transitions (k);

        if (k)
            define_states(k);
        else
            init_states();
    }
}
```

After setting the appropriate logic (QF_AUFBV), each individual step k is encoded iteratively, starting with the declaration of state variables.

```
void declare_states (uint k)
{
    // thread states
    declare_accu(k);
    declare_mem(k);
    declare_adr(k);
    declare_val(k);
    declare_full(k);
    declare_stmt(k);
    declare_block(k);
    declare_halt(k);

    // machine states
    declare_heap(k);
    declare_exit(k);

    if (!k) declare_exit_code();
}
```

Similarly to the generation of SMT-LIB commands, the following variadic function template serves as the basis for generating symbols, matching our established variable naming scheme.

```
template <class ... T>
std::string var (const std::string & name, const T & ... attributes)
{
    return (((name += ' ') += std::to_string(attributes)), ...);
}
```

This allows us to define a generator function for each variable as a simple wrapper, like the one of our accumulator register variables accu_t^k given below.

```
std::string accu (uint k, uint t) { return var("accu", k, t); }
```

The declaration of accumulator register states for each thread t in step k is now appended to the formula by using previously defined generator functions.

```
void declare_accu (uint k)
{
    for (uint t = 0; t < programs.size(); t++)
        formula << declare_bv(accu(k, t)) << '\n';
}
```

Example: `declare_accu(0)`

```
(declare-fun accu_0_0 () (_ BitVec 16))
(declare-fun accu_0_1 () (_ BitVec 16))
(declare-fun accu_0_2 () (_ BitVec 16))
```

While the declarator functions of the remaining register states mem_t^k , adr_t^k , val_t^k and full_t^k are defined almost identically, except for the latter changing the sort to `Bool`, an additional nested loop is needed to declare an activation variable for each statement in a thread's program.

```
void declare_stmt (uint k)
{
    for (uint t = 0; t < programs.size(); t++)
        for (uint pc = 0; pc < programs[t].size(); pc++)
            formula << declare_bool(stmt(k, t, pc)) << '\n';
}
```

Example: declare_stmt(0)

```
(declare-fun stmt_0_0_0 () Bool)
(declare-fun stmt_0_0_1 () Bool)
(declare-fun stmt_0_0_2 () Bool)
(declare-fun stmt_0_0_3 () Bool)
(declare-fun stmt_0_0_4 () Bool)

(declare-fun stmt_0_1_0 () Bool)
(declare-fun stmt_0_1_1 () Bool)
(declare-fun stmt_0_1_2 () Bool)
(declare-fun stmt_0_1_3 () Bool)
(declare-fun stmt_0_1_4 () Bool)

(declare-fun stmt_0_2_0 () Bool)
(declare-fun stmt_0_2_1 () Bool)
(declare-fun stmt_0_2_2 () Bool)
(declare-fun stmt_0_2_3 () Bool)
(declare-fun stmt_0_2_4 () Bool)
(declare-fun stmt_0_2_5 () Bool)
```

In order to avoid repeated iteration of input programs, the precomputed utility variables are used to indicate the presence of certain instructions and other control flow related information. Declaration of block flags $\text{block}_{id,t}^k$ can therefore be simplified by relying on the `checkpoints` map, containing the list of threads eventually waiting for a specific checkpoint.

```
void declare_block (uint k)
{
    for (const auto & [id, threads] : checkpoints)
        for (const auto & [t, _] : threads)
            formula << declare_bool(block(k, id, t)) << '\n';
}
```

Example: declare_block(0)

```
(declare-fun block_0_0_0 () Bool)
(declare-fun block_0_0_1 () Bool)
(declare-fun block_0_0_2 () Bool)
```

Halt flags halt_t^k are again declared by the appropriate generator function.

```
void declare_halt (uint k)
{
    for (uint t = 0; t < programs.size(); t++)
        formula << declare_bool(halt(k, t)) << '\n';
}
```

Example: `declare_halt(0)`

```
(declare-fun halt_0_0 () Bool)
(declare-fun halt_0_1 () Bool)
(declare-fun halt_0_2 () Bool)
```

Machine states heap^k , exit^k and exit-code^k are declared next. In contrast to our basic encoding scheme, only a single bit-vector variable representing the machine's final exit code is used and declared during the initial step. We omit the definition of the corresponding declarator functions as they just add a single variable with the appropriate sort.

Example: `declare_heap(0)`

```
(declare-fun heap_0 () (Array (_ BitVec 16) (_ BitVec 16)))
```

Example: `declare_exit(0).`

```
(declare-fun exit_0 () Bool)
```

Example: `declare_exit_code()`

```
(declare-fun exit-code () (_ BitVec 16))
```

Variable declaration is concluded by adding the required transition variables.

```
void declare_transitions (uint k)
{
    declare_thread(k);
    declare_flush(k);
    declare_exec(k);
    declare_check(k);
}
```

We skip further details about their declarator functions, as they are based on the same principles used for register states and statement activation variables, except check_{id}^k which also refers to the identifiers stored in the checkpoints map.


```

void declare_check (uint k)
{
    for (const auto & [id, _] : checkpoints)
        formula << declare_bool(check(k, id)) << '\n';
}

```

Example: declare_check(0)

```

(declare-fun check_0_0 () Bool)

```

With all variables being declared, the actual encoding process is started by defining helper variables and scheduling constraints according to our machine model.

```

void define_transitions (uint k)
{
    define_exec(k);
    define_check(k);
    define_cardinality_constraints(k);
    define_store_buffer_constraints(k);
    define_checkpoint_constraints(k);
    define_halt_constraints(k);
}

```

Execution variables $\text{exec}_{t,pc}^k$ are defined as a conjunction of the corresponding statement and thread activation variables.

```

void define_exec (uint k)
{
    for (uint t = 0; t < programs.size(); t++)
        for (uint pc = 0; pc < programs[t].size(); pc++)
            formula << assign(exec(k, t, pc),
                               land(stmt(k, t, pc),
                                     thread(k, t))) << '\n';
}

```

Example: define_exec(0)

```

(assert (= exec_0_0_0 (and stmt_0_0_0 thread_0_0)))
(assert (= exec_0_0_1 (and stmt_0_0_1 thread_0_0)))
(assert (= exec_0_0_2 (and stmt_0_0_2 thread_0_0)))
(assert (= exec_0_0_3 (and stmt_0_0_3 thread_0_0)))
(assert (= exec_0_0_4 (and stmt_0_0_4 thread_0_0)))

```

```

(assert (= exec_0_1_0 (and stmt_0_1_0 thread_0_1)))
(assert (= exec_0_1_1 (and stmt_0_1_1 thread_0_1)))
(assert (= exec_0_1_2 (and stmt_0_1_2 thread_0_1)))
(assert (= exec_0_1_3 (and stmt_0_1_3 thread_0_1)))
(assert (= exec_0_1_4 (and stmt_0_1_4 thread_0_1)))

(assert (= exec_0_2_0 (and stmt_0_2_0 thread_0_2)))
(assert (= exec_0_2_1 (and stmt_0_2_1 thread_0_2)))
(assert (= exec_0_2_2 (and stmt_0_2_2 thread_0_2)))
(assert (= exec_0_2_3 (and stmt_0_2_3 thread_0_2)))
(assert (= exec_0_2_4 (and stmt_0_2_4 thread_0_2)))
(assert (= exec_0_2_5 (and stmt_0_2_5 thread_0_2)))

```

Definition of checkpoint variables check_{id}^k is again based on the threads stored in checkpoints to generate the list of corresponding block variables $\text{block}_{id,t}^k$ for creating the required conjunction.

```

void define_check (uint k)
{
    for (const auto & [id, threads] : checkpoints)
    {
        std::vector<std::string> args;
        for (const auto & [t, _] : threads)
            args.push_back(block(k, id, t));
        formula << assign(check(k, id), land(args)) << '\n';
    }
}

```

Example: `define_check(0)`

```

(assert (= check_0_0 (and block_0_0_0 block_0_0_1 block_0_0_2)))

```

We continue with the cardinality constraint, based on the following functions implementing the different *at-most-one* predicates.

```

void cardinality_naive (const std::vector<std::string> & vars)
{
    for (auto i = vars.begin(); i != vars.end(); ++i)
        for (auto j = i + 1; j != vars.end(); ++j)
            formula << assertion(lor(lnot(*i), lnot(*j))) << '\n';
}

```

```

void cardinality_sequential (const std::vector<std::string> & vars)
{
    std::vector<std::string> auxs;
    const auto end = --vars.end();
    for (auto i = vars.begin(); i != end; ++i)
        formula << declare_bool(auxs.emplace_back(*i + "_aux")) << '\n';
    auto var = vars.begin();
    auto aux = auxs.begin();
    formula << assertion(lor(lnot(*var), *aux)) << '\n';
    while (++var != end)
    {
        const std::string & prev = *aux++;
        formula << assertion(lor(lnot(*var), *aux)) << '\n'
            << assertion(lor(lnot(prev), *aux)) << '\n'
            << assertion(lor(lnot(*var), lnot(prev))) << '\n';
    }
    formula << assertion(lor(lnot(*var), lnot(*aux))) << '\n';
}

```

The required *exactly-one* constraint can now be defined by collecting the relevant variables and selecting the appropriate *at-most-one* predicate after adding a disjunction expressing that *at-least-one* needs to be true.

```

void define_cardinality_constraints (uint k)
{
    std::vector<std::string> vars;
    for (uint t = 0; t < programs.size(); t++)
    {
        vars.push_back(thread(k, t));
        vars.push_back(flush(k, t));
    }
    vars.push_back(exit(k));
    // >= 1 constraint
    formula << assertion(lor(vars)) << '\n';
    // <= 1 constraint
    if (vars.size() > 5)
        cardinality_sequential(vars);
    else
        cardinality_naive(vars);
}

```

Example: `define_cardinality_constraints(0)`

```
(assert (or thread_0_0 flush_0_0
            thread_0_1 flush_0_1
            thread_0_2 flush_0_2
            exit_0))

(declare-fun thread_0_0_aux () Bool)
(declare-fun flush_0_0_aux () Bool)
(declare-fun thread_0_1_aux () Bool)
(declare-fun flush_0_1_aux () Bool)
(declare-fun thread_0_2_aux () Bool)
(declare-fun flush_0_2_aux () Bool)

(assert (or (not thread_0_0) thread_0_0_aux))
(assert (or (not flush_0_0) flush_0_0_aux))
(assert (or (not thread_0_0_aux) flush_0_0_aux))
(assert (or (not flush_0_0) (not thread_0_0_aux)))
(assert (or (not thread_0_1) thread_0_1_aux))
(assert (or (not flush_0_0_aux) thread_0_1_aux))
(assert (or (not thread_0_1) (not flush_0_0_aux)))
(assert (or (not flush_0_1) flush_0_1_aux))
(assert (or (not thread_0_1_aux) flush_0_1_aux))
(assert (or (not flush_0_1) (not thread_0_1_aux)))
(assert (or (not thread_0_2) thread_0_2_aux))
(assert (or (not flush_0_1_aux) thread_0_2_aux))
(assert (or (not thread_0_2) (not flush_0_1_aux)))
(assert (or (not flush_0_2) flush_0_2_aux))
(assert (or (not thread_0_2_aux) flush_0_2_aux))
(assert (or (not flush_0_2) (not thread_0_2_aux)))
(assert (or (not exit_0) (not flush_0_2_aux)))
```

Since a program might not even contain a single memory barrier, different store buffer related constraints are added for each thread. Beside the common restriction of flushing an empty store buffer, we resort to **barriers** containing the program counters of barrier instructions for generating the list of corresponding statement activation variables required to additionally delay their execution while the store buffer is full.

```

void define_store_buffer_constraints (uint k)
{
    for (uint t = 0; t < programs.size(); t++)
        if (barriers[t].empty())
            formula << assertion(implies(flush(k, t), full(k, t))) << '\n';
        else
        {
            std::vector<std::string> stmts;
            for (uint pc : barriers[t])
                stmts.push_back(stmt(k, t, pc));
            formula << assertion(ite(full(k, t),
                                    implies(lor(stmts),
                                              lnot(thread(k, t))),
                                    lnot(flush(k, t)))) << '\n';
        }
}

```

Example: define_store_buffer_constraints(0)

```

(assert (ite full_0_0
            (=> (or stmt_0_0_1 stmt_0_0_4) (not thread_0_0))
            (not flush_0_0)))
(assert (ite full_0_1
            (=> (or stmt_0_1_1 stmt_0_1_4) (not thread_0_1))
            (not flush_0_1)))
(assert (=> flush_0_2 full_0_2))

```

Checkpoint constraints are defined by using checkpoints to generate the relevant $\text{block}_{id,t}^k$ and check_{id}^k variables for explicitly disabling a thread's activation while it is waiting for all other participants to synchronize upon checkpoint id .

```

void define_checkpoint_constraints (uint k)
{
    for (const auto & [id, threads] : checkpoints)
        for (const auto & [t, _] : threads)
            formula << assertion(implies(land(block(k, id, t),
                                                lnot(check(k, id))),
                                         lnot(thread(k, t)))) << '\n';
}

```

Example: `define_checkpoint_constraints(0)`

```
(assert (=> (and block_0_0_0 (not check_0_0)) (not thread_0_0)))
(assert (=> (and block_0_0_1 (not check_0_0)) (not thread_0_1)))
(assert (=> (and block_0_0_2 (not check_0_0)) (not thread_0_2)))
```

The definition of scheduling constraints is concluded by preventing halted threads from being executed.

```
void define_halt_constraints (uint k)
{
    for (uint t = 0; t < programs.size(); t++)
        formula << assertion(implies(halt(k, t),
                                      lnot(thread(k, t)))) << '\n';
}
```

Example: `define_halt_constraints(0)`

```
(assert (=> halt_0_0 (not thread_0_0)))
(assert (=> halt_0_1 (not thread_0_1)))
(assert (=> halt_0_2 (not thread_0_2)))
```

Encoding of each particular step ends with the definition of machine states, starting with their initialization in the first step $k = 0$.

```
void init_states ()
{
    init_accu();
    init_mem();
    init_adr();
    init_val();
    init_full();
    init_stmt();
    init_block();
    init_halt();
    init_heap();
    init_exit();
}
```

We begin by setting each thread's initial accumulator register state to zero.

```
void init_accu ()
{
    for (uint t = 0; t < programs.size(); t++)
        formula << assign(accu(0, t), consth(0)) << '\n';
}
```

Example: `init_accu()`

```
(assert (= accu_0_0 #x0000))
(assert (= accu_0_1 #x0000))
(assert (= accu_0_2 #x0000))
```

Similar to their declaration, we skip the initialization functions of the remaining register states as they are again defined almost identical and continue with the activation of every thread's initial statement $\text{stmt}_{t,0}^0$.

```
void init_stmt ()
{
    for (uint t = 0; t < programs.size(); t++)
        for (uint pc = 0; pc < programs[t].size(); pc++)
            formula << assertion(pc ? lnot(stmt(0, t, pc))
                                : stmt(0, t, pc)) << '\n';
}
```

Example: `init_stmt()`

```
(assert stmt_0_0_0)
(assert (not stmt_0_0_1))
(assert (not stmt_0_0_2))
(assert (not stmt_0_0_3))
(assert (not stmt_0_0_4))

(assert stmt_0_1_0)
(assert (not stmt_0_1_1))
(assert (not stmt_0_1_2))
(assert (not stmt_0_1_3))
(assert (not stmt_0_1_4))

(assert stmt_0_2_0)
(assert (not stmt_0_2_1))
(assert (not stmt_0_2_2))
(assert (not stmt_0_2_3))
(assert (not stmt_0_2_4))
(assert (not stmt_0_2_5))
```

Block flags $\text{block}_{id,t}^0$ are initially disabled by relying on the entries given in the checkpoints map.

```

void init_block ()
{
    for (const auto & [id, threads] : checkpoints)
        for (const auto & [t, _] : threads)
            formula << assertion(lnot(block(0, id, t))) << '\n';
}

```

Example: init_block()

```

(assert (not block_0_0_0))
(assert (not block_0_0_1))
(assert (not block_0_0_2))

```

Halt flags halt_t^0 of each thread are initialized in a similar manner.

```

void init_halt ()
{
    for (uint t = 0; t < programs.size(); t++)
        formula << assertion(lnot(halt(0, t))) << '\n';
}

```

Example: init_halt()

```

(assert (not halt_0_0))
(assert (not halt_0_1))
(assert (not halt_0_2))

```

Although the machine's memory is considered to be uninitialized in general, input data is assigned according to the entries of a given memory map.

```

void init_heap ()
{
    for (const auto & [adr, val] : mmap)
        formula << assign(select(heap(0), consth(adr)),
                           consth(val)) << '\n';
}

```

Example: init_heap()

```

(assert (= (select heap_0 #x0000) #x0000))
(assert (= (select heap_0 #x0001) #x0000))

```

Finally, the initial exit flag exit_0 is disabled to enforce execution.


```

void init_exit()
{
    formula << assertion(!not(exit(0))) << '\n';
}

```

Example: `init_exit()`

```

(assert (not exit_0))

```

Memory access according to the previously defined load_t^k is implemented by the following helper function.

```

std::string load (uint k, uint t, uint adr, bool indirect = false)
{
    std::string address = consth(adr);
    std::string adr_var = adr(k, t);
    std::string val_var = val(k, t);
    std::string full_var = full(k, t);
    std::string heap_var = heap(k);

    if (indirect)
        return
            ite(full_var,
                ite(equal(adr_var, address),
                    ite(equal(val_var, address),
                        val_var,
                        select(heap_var, val_var)),
                    ite(equal(adr_var, select(heap_var, address)),
                        val_var,
                        select(heap_var, select(heap_var, address))))),
                select(heap_var, select(heap_var, address)));
    else
        return
            ite(land(full_var, equal(adr_var, address)),
                val_var,
                select(heap_var, address));
}

```

Common encoding functions are concluded by introducing implementations for `Instruction::encode(uint, uint, State)`, returning the successor of a given state after executing the specific instruction expect for the statement activation variables $\text{stmt}_{t,pc}^k$, which are handled separately and only jump conditions being generated this way. Implementations for `FENCE`, `JMP` and `HALT` instructions are omitted, as they just return an empty `std::string`.

LOAD arg

```
std::string Load::encode (uint k, uint t, State state)
{
    return load(k, t, arg, indirect);
}
```

STORE arg

```
std::string Store::encode (uint k, uint t, State state)
{
    switch (state)
    {
        case State::adr: return indirect ? load(k, arg) : consth(arg);
        case State::val: return accu(k, t);
    }
}
```

ADD arg

```
std::string Add::encode (uint k, uint t, State state)
{
    return bvadd(accu(k, t), load(k, arg, indirect));
}
```

ADDI arg

```
std::string Addi::encode (uint k, uint t, State state)
{
    return bvadd(accu(k, t), consth(arg));
}
```

SUB arg

```
std::string Sub::encode (uint k, uint t, State state)
{
    return bvsub(accu(k, t), load(k, arg, indirect));
}
```

SUBI arg

```
std::string Subi::encode (uint k, uint t, State state)
{
    return bvsub(accu(k, t), consth(arg));
}
```

MUL arg

```
std::string Mul::encode (uint k, uint t, State state)
{
    return bvmul(accum(k, t), load(k, arg, indirect));
}
```

MULI arg

```
std::string Muli::encode (uint k, uint t, State state)
{
    return bvmul(accum(k, t), consth(arg));
}
```

CMP arg

```
std::string Cmp::encode (uint k, uint t, State state)
{
    return bvsub(accum(k, t), load(k, arg, indirect));
}
```

JZ arg

```
std::string Jz::encode (uint k, uint t, State state)
{
    return equal(accum(k, t), consth(0));
}
```

JNZ arg

```
std::string Jnz::encode (uint k, uint t, State state)
{
    return lnot(equal(accum(k, t), consth(0)));
}
```

JS arg

```
std::string Js::encode (uint k, uint t, State state)
{
    return equal("#b1", extract("15", "15", accum(k, t)));
}
```

JNS arg

```
std::string Jns::encode (uint k, uint t, State state)
{
    return equal("#b0", extract("15", "15", accu(k, t)));
}
```

JNZNS arg

```
std::string Jnzns::encode (uint k, uint t, State state)
{
    return land(lnot(equal(accu(k, t), consth(0))),
               equal("#b0", extract("15", "15", accu(k, t))));
}
```

MEM arg

```
std::string Mem::encode (uint k, uint t, State state)
{
    return load(k, arg, indirect);
}
```

CAS arg

```
std::string Cas::encode (uint k, uint t, State state)
{
    auto heap_var = heap(k);
    auto address = indirect ? select(heap_var, consth(arg))
                             : consth(arg);
    auto condition = equal(mem(k, t), select(heap_var, address));
    switch (state)
    {
        case State::accu: return ite(condition, consth(1), consth(0));
        case State::heap: return ite(condition,
                                       store(heap_var,
                                             address,
                                             accu(k, t)),
                                       heap_var);
    }
}
```

EXIT arg

```
std::string Exit::encode (uint k, uint t, State state)
{
    return consth(arg);
}
```

6 Functional Next State Logic

In principle there are two ways to encode state updates: a functional, relying on `ite` cascades to perform *static single assignments* for determining the successor of each state explicitly and a relational, based on implying the next state for each possible transition. While the relational version is easier to encode since it is closer to our model’s semantics and results in a simpler structure, the functional encoding has the benefit of being more compact and also turned out to be more efficient for SMT solving. We will now introduce our functional state update scheme, generated by using the `-e smtlib` command line parameter.

```
void define_states (uint k)
{
    define_accu(k);
    define_mem(k);
    define_adr(k);
    define_val(k);
    define_full(k);
    define_stmt(k);
    define_block(k);
    define_halt(k);
    define_heap(k);
    define_exit(k);

    if (k == bound)
        define_exit_code();
}
```

Starting with the accumulator registers accu_t^k , each update expression is initialized with its predecessor accu_t^{k-1} , forming the base case of preserving the previous state. Any program statement altering the accumulator then extends the expression by embedding its current value in the *else* branch of an *ite*, using the corresponding instruction’s `encode` implementation to set the successor state depending on the execution variable $\text{exec}_{t,pc}^{k-1}$.

```

void define_accu (uint k)
{
  for (uint t = 0; t < programs.size(); t++)
  {
    std::string next = accu(k - 1, t);
    const auto & stmts = updates[State::accu][t];
    for (auto pc = stmts.rbegin(); pc != stmts.rend(); ++pc)
      next = ite(exec(k - 1, t, *pc),
                 program[t][*pc]->encode(k - 1, t, State::accu),
                 next);
    formula << assign(accu(k, t), next) << '\n';
  }
}

```

Example: define_accu(1)

```

(assert (= accu_1_0
  (ite exec_0_0_0
    (bvadd accu_0_0 #x0001)
    (ite exec_0_0_2
      (ite (and full_0_0 (= adr_0_0 #x0001))
        val_0_0
        (select heap_0 #x0001))
      accu_0_0))))
(assert (= accu_1_1 ... ; same as before but reading address 0
(assert (= accu_1_2
  (ite exec_0_2_1
    (bvadd accu_0_2
      (ite (and full_0_2 (= adr_0_2 #x0000))
        val_0_2
        (select heap_0 #x0000)))
    (ite exec_0_2_2
      (bvadd accu_0_2
        (ite (and full_0_2
          (= adr_0_2 #x0001))
          val_0_2
          (select heap_0 #x0001)))
        accu_0_2))))

```

We omit the implementation of state update functions for the remaining register states mem_t^k , adr_t^k and val_t^k as they only differ in using the appropriate `State` entries and variable generators, but include their output to show the state updates generated for our demo example.

Example: `define_mem(1)`

```
(assert (= mem_1_0 mem_0_0))
(assert (= mem_1_1 mem_0_1))
(assert (= mem_1_2 mem_0_2))
```

Example: `define_adr(1)`

```
(assert (= adr_1_0 (ite exec_0_0_1 #x0000 adr_0_0)))
(assert (= adr_1_1 (ite exec_0_1_1 #x0001 adr_0_1)))
(assert (= adr_1_2 adr_0_2))
```

Example: `define_val(1)`

```
(assert (= val_1_0 (ite exec_0_0_1 accu_0_0 val_0_0)))
(assert (= val_1_1 (ite exec_0_1_1 accu_0_1 val_0_1)))
(assert (= val_1_2 val_0_2))
```

Store buffer full flags full_t^k are defined by a single `ite`, either falsifying the state in case the store buffer has been flushed, or assigning a conjunction over all execution variables $\text{exec}_{t,pc}^{k-1}$ of `STORE` instructions to set it if an entry was added, as well as the predecessor full_t^{k-1} for preserving its state otherwise.

```
void define_full (uint k)
{
  for (uint t = 0; t < programs.size(); t++)
  {
    std::vector<std::string> args {full(k - 1, t)};
    for (uint pc : updates[State::full][t])
      args.push_back(exec(k - 1, t, pc));
    formula << assign(full(k, t),
                      ite(flush(k - 1, t),
                          "false",
                          lor(args))) << '\n';
  }
}
```

Example: `define_full(1)`

```
(assert (= full_1_0 (ite flush_0_0
                          false
                          (or full_0_0 exec_0_0_1))))
(assert (= full_1_1 (ite flush_0_1
                          false
                          (or full_0_1 exec_0_1_1))))
(assert (= full_1_2 (ite flush_0_2 false full_0_2)))
```

In order to correctly model symbolic program counters correctly, individual state updates must ensure that at most one statement is activated in any step. We do so by initializing the update expression of each statement activation variable $\text{stmt}_{t,pc}^k$ with a conjunction containing its previous activation and the negation of the corresponding execution variable $\text{exec}_{t,pc}^{k-1}$ for explicitly deactivating the statement if it has been executed while preserving its state otherwise. Similarly to defining the successor of register states, each preceding program statement then embeds the expression's current value in the *else* branch of an *ite*, activating the statement depending on the predecessor's execution. Special care must be taken in case of the predecessor being a conditional jump, identified by *is_jump*, to either activate the target using a conjunction of its execution variable and condition, generated by the relevant *encode* implementation, or the next statement by negating the condition in case of a failed jump.

```

void define_stmt (uint k)
{
  for (uint t = 0; t < programs.size(); t++)
    for (uint pc = 0; pc < programs[t].size(); pc++)
    {
      // statement reactivation
      std::string next = land(stmt(k - 1, t, pc),
                           lnot(exec(k - 1, t, pc)));
      // activation by predecessor
      const auto & stmts = predecessors[t][pc];
      for (auto pre = stmts.rbegin(); pre != stmts.rend(); ++pre)
      {
        std::string val = exec(k - 1, t, *pre);
        Instruction * op = programs[t][*pre];
        // add condition if predecessor is a jump
        if (is_jump(op))
        {
          std::string cond = op->encode(k - 1, t, State::stmt);
          val = land(val, op->arg == pc ? cond : lnot(cond));
        }
        next = ite(stmt(k - 1, t, *pre), val, next);
      }
      formula << assign(stmt(k, t, pc), next) << '\n';
    }
}

```

Example: define_stmt(1)


```

(assert (= stmt_1_0_0 (and stmt_0_0_0 (not exec_0_0_0))))
(assert (= stmt_1_0_1 (ite stmt_0_0_0
                           exec_0_0_0
                           (and stmt_0_0_1 (not exec_0_0_1)))))
(assert (= stmt_1_0_2 (ite stmt_0_0_1
                           exec_0_0_1
                           (and stmt_0_0_2 (not exec_0_0_2)))))
(assert (= stmt_1_0_3 (ite stmt_0_0_2
                           exec_0_0_2
                           (and stmt_0_0_3 (not exec_0_0_3)))))
(assert (= stmt_1_0_4 (ite stmt_0_0_3
                           exec_0_0_3
                           (and stmt_0_0_4 (not exec_0_0_4)))))
(assert (= stmt_1_1_0 (and stmt_0_1_0 (not exec_0_1_0))))
(assert (= stmt_1_1_1 (ite stmt_0_1_0
                           exec_0_1_0
                           (and stmt_0_1_1 (not exec_0_1_1)))))
(assert (= stmt_1_1_2 (ite stmt_0_1_1
                           exec_0_1_1
                           (and stmt_0_1_2 (not exec_0_1_2)))))
(assert (= stmt_1_1_3 (ite stmt_0_1_2
                           exec_0_1_2
                           (and stmt_0_1_3 (not exec_0_1_3)))))
(assert (= stmt_1_1_4 (ite stmt_0_1_3
                           exec_0_1_3
                           (and stmt_0_1_4 (not exec_0_1_4)))))
(assert (= stmt_1_2_0 (and stmt_0_2_0 (not exec_0_2_0))))
(assert (= stmt_1_2_1 (ite stmt_0_2_0
                           exec_0_2_0
                           (and stmt_0_2_1 (not exec_0_2_1)))))
(assert (= stmt_1_2_2 (ite stmt_0_2_1
                           exec_0_2_1
                           (and stmt_0_2_2 (not exec_0_2_2)))))
(assert (= stmt_1_2_3 (ite stmt_0_2_2
                           exec_0_2_2
                           (and stmt_0_2_3 (not exec_0_2_3)))))
(assert (= stmt_1_2_4 (ite stmt_0_2_3
                           (and exec_0_2_3 (not (= accu_0_2
                                                    #x0000))))
                           (and stmt_0_2_4 (not exec_0_2_4)))))
(assert (= stmt_1_2_5 (ite stmt_0_2_3
                           (and exec_0_2_3 (= accu_0_2 #x0000))
                           (and stmt_0_2_5 (not exec_0_2_5)))))

```

Block flags $\text{block}_{id,t}^k$ are defined by a single `ite`, unblocking the thread by resetting the state if all threads synchronized upon the corresponding checkpoint id in the previous step. Otherwise, we assign a disjunction including all execution variables $\text{exec}_{t,pc}^{k-1}$ of related `CHECK` id instructions enabling the state as well as the predecessor $\text{block}_{id,t}^{k-1}$ to preserve its value.

```
void define_block (uint k)
{
  for (const auto & [id, threads] : checkpoints)
    for (const auto & [t, stmts] : threads)
      {
        std::vector<std::string> args {block(k - 1, id, t)};
        for (uint pc : stmts)
          args.push_back(exec(k - 1, t, pc));
        formula << assign(block(k, id, t),
                          ite(check(k - 1, id),
                              "false",
                              lor(args))) << '\n';
      }
}
```

Example: `define_block(1)`

```
(assert (= block_1_0_0 (ite check_0_0
                             false
                             (or block_0_0_0 exec_0_0_3))))
(assert (= block_1_0_1 (ite check_0_0
                             false
                             (or block_0_0_1 exec_0_1_3))))
(assert (= block_1_0_2 (ite check_0_0
                             false
                             (or block_0_0_2 exec_0_2_0))))
```

Since halt flags halt_t^k cannot be reset once a thread has been stopped, they are defined by a simple disjunction including the execution variables $\text{exec}_{t,pc}^{k-1}$ of `HALT` instructions and the previous state halt_t^{k-1} to preserve their value.

```

void define_halt (uint k)
{
  for (uint t = 0; t < programs.size(); t++)
  {
    std::vector<std::string> args {halt(k - 1, t)};
    for (uint pc : updates[State::halt][t])
      args.push_back(exec(k - 1, t, pc));
    formula << assign(halt(k, t), lor(args)) << '\n';
  }
}

```

Example: define_halt(1)

```

(assert (= halt_1_0 (or halt_0_0 exec_0_0_4)))
(assert (= halt_1_1 (or halt_0_1 exec_0_1_4)))
(assert (= halt_1_2 halt_0_2))

```

Definition of our shared memory array heap^k again starts by initializing the update expression with its previous state heap^{k-1} . Each thread then extends the expression by embedding it in an `ite` for every atomic write, modifying the array according to the instruction's `encode` implementation, followed by a final if-then-else to store the store buffer entry in case it was flushed.

```

void define_heap (uint k)
{
  std::string next = heap(k - 1);
  for (int t = programs.size() - 1; t >= 0; t--)
  {
    const auto & stmts = updates[State::heap][t];
    for (auto pc = stmts.rbegin(); pc != stmts.rend(); ++pc)
      next = ite(exec(k - 1, t, *pc),
                 programs[t][*pc].encode(k - 1, t, State::heap),
                 next);
    next = ite(flush(k - 1, t),
               store(heap(k - 1),
                     adr(k - 1, t),
                     val(k - 1, t)),
               next);
  }
  formula << assign(heap(k, t), next) << '\n';
}

```

Example: `define_heap(1)`

```
(assert (= heap_1
  (ite flush_0_0
    (store heap_0 adr_0_0 val_0_0)
    (ite flush_0_1
      (store heap_0 adr_0_1 val_0_1)
      (ite flush_0_2
        (store heap_0 adr_0_2 val_0_2)
        heap_0))))))
```

In order to enable the exit flag exit^k according to our termination criteria, we define it as a disjunction containing the execution variables $\text{exec}_{t,pc}^{k-1}$ of every `EXIT` statement, a conjunction over the halt variables halt_t^k of threads containing a call to `HALT` for stopping the machine if no more threads are running and the previous state exit^{k-1} to preserve its value.

```
void define_exit(uint k)
{
  std::vector<std::string> halts;
  for (const auto & stmts : updates[State::halt])
    for (uint pc : stmts)
      halts.push_back(halt(k, t));
  std::vector<std::string> args {exit(k - 1), land(halts)};
  for (const auto & stmts : updates[State::exit])
    for (uint pc : stmts)
      args.push_back(exec(k - 1, t, pc));
  formula << assign(exit(k), lor(args)) << '\n';
}
```

Example: `define_exit(1)`

```
(assert (= exit_1
  (or exit_0
    (and halt_1_0 halt_1_1)
    exec_0_2_4
    exec_0_2_5)))
```

Our functional encoding scheme is concluded by setting the machine's exit code during the final step. Since it is unique in every execution, explicit unrolling allows us to reduce the total number of variables by introducing only a single state `exit-code`, defined by an expression initialized with zero and embedded in an `ite` for each step $k \in [0, \text{bound}]$ and `EXIT` statement, assigning the corresponding exit code in case of its execution.

```

void define_exit_code ()
{
    std::string next = consth(0);
    for (uint k = 0; k <= bound; k++)
        for (const auto & stmts : updates[State::exit])
            for (uint pc : stmts)
                next = ite(exec(k, t, pc),
                           programs[t][pc].encode(k, t, State::exit),
                           next);
    formula << assign(exit_code, next) << '\n';
}

```

```

(assert (= exit-code
            (ite exec_17_2_5
                  #x0001
                  (ite exec_17_2_4
                        #x0000
                        ...
                        (ite exec_0_2_5
                              #x0001
                              (ite exec_0_2_4
                                    #x0000
                                    #x0000)))))))

```

7 Relational Next State Logic

Since state of the art SAT and SMT solvers are able to exploit certain structures in the problem formulation, the relative hardness of different semantically equivalent encodings may vary quite drastically. To highlight these differences, we included another variant using a relational next state logic, generated with the `-e smtlib-relational` command line parameter, based on implying the successors for each possible transition. Although this flat encoding scheme tends to work well with large combinatorial problems and small domains, our experiments show that it requires considerably more time to solve compared to our functional approach.

```
void define_states (uint k)
{
    for (uint t = 0; t < programs.size(); t++)
    {
        imply_thread_executed(k, t);
        imply_thread_not_executed(k, t);
        imply_thread_flushed(k, t);
    }
    imply_machine_exited(k);
}
```

We begin by defining variadic template functions for simplifying the generation of frequently reoccurring expressions, used to restore a state's previous value and additionally reset flags depending on a given condition.

```
template <class R, class ... T>
R restore (R (*var) (uint k, T ... args), uint k, T ... args)
{
    return equal(var(k, args...), var(k - 1, args...));
}

template <class R, class ... T>
R reset (R (*var) (uint k, uint x, T ... args),
         R (*cond) (uint k, uint x),
         uint k, uint x, T ... args)
{
    return equal(var(k, x, args...),
                 ite(cond(k - 1, x),
                     "false",
                     var(k - 1, x, args...)));
}
```

Implying a common set of states that might be influenced by executing a certain thread results in most of them remaining unchanged. In order to reduce the effort involved in implementing the encoding function of each instruction, we introduce the following map, using the previously defined template functions to initialize it with the corresponding state-preserving expressions, thus requiring only the entries of states which are actually altered to be replaced and provide an implicit `std::string` conversion operator returning a conjunction to define all successors at once.

```
struct Next : std::map<State, std::string>
{
    Next (k, t)
    {
        (*this)[State::accu] = restore(&accu, k, t);
        (*this)[State::mem] = restore(&mem, k, t);
        (*this)[State::adr] = restore(&adr, k, t);
        (*this)[State::val] = restore(&val, k, t);
        (*this)[State::full] = restore(&full, k, t);
        (*this)[State::halt] = restore(&halt, k, t);
        (*this)[State::heap] = restore(&heap, k);
        (*this)[State::exit] = lnot(exit(k, t));
        std::vector<std::string> block_vars;
        for (const auto & [id, threads] : checkpoints)
            if (threads.find(t) != threads.end())
                block_vars.push_back(reset(&block,
                                           &check,
                                           k, id, t));
        if (!block_vars.empty())
            (*this)[State::block] = land(block_vars);
    }

    operator std::string () const
    {
        std::vector<std::string> args;
        for (const auto & [_, expr] : *this)
            if (!expr.empty())
                args.push_back(expr);
        return land(args);
    }
};
```

Since the statement activation variables $\text{stmt}_{t,pc}^k$ of each thread must be fully defined for every possible transition, we include helper functions to generate the required conjunctions.

```
std::string activate (uint k, uint t, uint pc)
{
    std::vector<std::string> stmts;
    for (uint i = 0; i < programs[t].size(); i++)
        stmts.push_back(i == pc ? stmt(k, t, pc)
                        : lnot(stmt(k, t, i)));
    return land(stmts);
}
```

In case of jump instructions, another overload either activates the target or successor statement, depending on the corresponding condition.

```
std::string activate (uint k, uint t, uint pc, Instruction * jmp)
{
    std::vector<std::string> stmts;
    std::string condition = jmp->encode(k - 1, t, State::stmt);
    for (uint i = 0; i < programs[t].size(); i++)
    {
        std::string stmt_var = stmt(k, t, i);
        if (i == jmp->arg)
            stmts.push_back(ite(condition, stmt_var, lnot(stmt_var)));
        else if (i == pc + 1)
            stmts.push_back(ite(condition, lnot(stmt_var), stmt_var));
        else
            stmts.push_back(lnot(stmt_var));
    }
    return land(stmts);
}
```

Generating a common set of successor states for every possible instruction is again based on dynamically dispatched virtual member functions of concrete `Instruction` class instantiations. Therefore, we introduce the following overload including the current program counter for activating the next statement.

```
virtual std::string Instruction::encode (uint t, uint k, uint pc) = 0;
```

By using the previously defined helper constructs, the actual implementations can be simplified quite drastically and even though they therefore almost look the same, we include the full definitions for completeness.

LOAD arg

```
std::string Load::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::accu] = equal(accum(k, t),
                             encode(k - 1, t, State::accu));
    next[State::stmt] = activate(k, t, pc + 1);
    return next;
}
```

STORE arg

```
std::string Store::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::adr] = equal(adr(k, t), encode(k - 1, t, State::adr));
    next[State::val] = equal(val(k, t), encode(k - 1, t, State::val));
    next[State::full] = full(k, t);
    next[State::stmt] = activate(k, t, pc + 1);
    return next;
}
```

FENCE

```
std::string Fence::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::stmt] = activate(k, t, pc + 1);
    return next;
}
```

ADD arg

```
std::string Add::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::accu] = equal(accum(k, t),
                             encode(k - 1, t, State::accu));
    next[State::stmt] = activate(k, t, pc + 1);
    return next;
}
```

ADDI arg

```
std::string Addi::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::accu] = equal(accu(k, t),
                             encode(k - 1, t, State::accu));
    next[State::stmt] = activate(k, t, pc + 1);
    return next;
}
```

SUB arg

```
std::string Sub::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::accu] = equal(accu(k, t),
                             encode(k - 1, t, State::accu));
    next[State::stmt] = activate(k, t, pc + 1);
    return next;
}
```

SUBI arg

```
std::string Subi::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::accu] = equal(accu(k, t),
                             encode(k - 1, t, State::accu));
    next[State::stmt] = activate(k, t, pc + 1);
    return next;
}
```

MUL arg

```
std::string Mul::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::accu] = equal(accu(k, t),
                             encode(k - 1, t, State::accu));
    next[State::stmt] = activate(k, t, pc + 1);
    return next;
}
```

MULI arg

```
std::string Muli::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::accu] = equal(accum(k, t),
                             encode(k - 1, t, State::accu));
    next[State::stmt] = activate(k, t, pc + 1);
    return next;
}
```

CMP arg

```
std::string Cmp::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::accu] = equal(accum(k, t),
                             encode(k - 1, t, State::accu));
    next[State::stmt] = activate(k, t, pc + 1);
    return next;
}
```

JMP arg

```
std::string Jmp::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::stmt] = activate(k, t, arg);
    return next;
}
```

JZ arg

```
std::string Jz::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::stmt] = activate(k, t, pc, this);
    return next;
}
```

JNZ arg

```
std::string Jnz::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::stmt] = activate(k, t, pc, this);
    return next;
}
```

JS arg

```
std::string Js::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::stmt] = activate(k, t, pc, this);
    return next;
}
```

JNS arg

```
std::string Jns::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::stmt] = activate(k, t, pc, this);
    return next;
}
```

JNZNS arg

```
std::string Jnzns::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::stmt] = activate(k, t, pc, this);
    return next;
}
```

MEM arg

```
std::string Mem::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::accu] = equal(accu(k, t),
                             encode(k - 1, t, State::accu));
    next[State::mem] = equal(mem(k, t), encode(k - 1, t, State::mem));
    next[State::stmt] = activate(k, t, pc + 1);
    return next;
}
```

CAS arg

```
std::string Cas::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::accu] = equal(accu(k, t),
                             encode(k - 1, t, State::accu));
    next[State::stmt] = activate(k, t, pc + 1);
    next[State::heap] = equal(heap(k),
                              encode(k - 1, t, State::heap));
    return next;
}
```

CHECK arg

```
std::string Check::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::stmt] = activate(k, t, pc + 1);
    std::vector<std::string> blocks;
    for (const auto & [id, threads] : checkpoints)
        if (threads.find(t) != threads.end())
            blocks.push_back(id == op.arg
                             ? block(k, id, t)
                             : reset(&block, &check, k, id, t));
    next[State::block] = land(blocks);
    return state;
}
```

Since a thread might wait for different checkpoints, the encode function has to enable the corresponding blocking flag $\text{block}_{id,t}^k$ while preserving or resetting any other.

HALT

```
std::string Halt::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::stmt] = activate(k, t, -1);
    if (programs.size() > 1)
    {
        std::vector<std::string> args;
        for (uint thread = 0; thread < programs.size(); thread++)
            if (thread != t)
                args.push_back(halt(k, thread));
        next[State::halt] =
            land(halt(k, t),
                ite(land(args),
                    land(exit(k), equal(exit_code, consth(0))),
                    lnot(exit(k))));
    }
    else
        next[State::halt] = land(halt(k, t),
                                exit(k),
                                equal(exit_code, consth(0)));
    state.erase(state.find(State::exit));
    return state;
}
```

Stopping the machine if all threads halted is achieved by an `ite`, enabling the exit flag `exitk` and assigning the default exit code if all halt flags `halttk` are set. Otherwise, the exit flag is disabled to continue execution. Since the exit flag is now already contained in the `State::halt` entry, we have to erase `State::exit` to prevent conflicts due to its falsification.

EXIT arg

```
std::string Exit::encode (uint t, uint k, uint pc)
{
    Next next(k, t);
    next[State::stmt] = activate(k, t, -1);
    next[State::exit] = exit(k);
    next[State::exit_code] = equal(exit_code,
                                   encode(t, k, State::exit_code));
    return state;
}
```

With all utility constructs and instruction related encoding functions defined, we are now able to imply the next states for all transitions, starting with the execution of any given statement.

```
void imply_thread_executed (uint k, uint t)
{
    for (uint pc = 0; pc < programs[t].size(); pc++)
        formula << assertion(imply(exec(k - 1, t, pc),
                                   programs[t][pc]->encode(k, t, pc)))
        << '\n';
}
```

Example: `imply_thread_executed(1, 0)`

```
(assert (=> exec_0_0_0 ; ADDI 1
    (and (= accu_1_0 (bvadd accu_0_0 #x0001))
        (= mem_1_0 mem_0_0)
        (= adr_1_0 adr_0_0)
        (= val_1_0 val_0_0)
        (= full_1_0 full_0_0)
        (and (not stmt_1_0_0)
            stmt_1_0_1
            (not stmt_1_0_2)
            (not stmt_1_0_3)
            (not stmt_1_0_4))
        (= block_1_0_0 (ite check_0_0 false block_0_0_0))
        (= halt_1_0 halt_0_0)
        (= heap_1 heap_0)
        (not exit_1))))
(assert (=> exec_0_0_1 ; STORE 0
    (and (= accu_1_0 accu_0_0)
        (= mem_1_0 mem_0_0)
        (= adr_1_0 #x0000)
        (= val_1_0 accu_0_0)
        full_1_0
        (and (not stmt_1_0_0)
            (not stmt_1_0_1)
            stmt_1_0_2
            (not stmt_1_0_3)
            (not stmt_1_0_4))
        (= block_1_0_0 (ite check_0_0 false block_0_0_0))
        (= halt_1_0 halt_0_0)
        (= heap_1 heap_0)
        (not exit_1)))) ...
```

In case a thread was not executed in the previous step, all local states must be preserved except the store buffer full flag full_t^k and block flags $\text{block}_{id,t}^k$, as they might be reset by a store buffer flush, or due to being released from a checkpoint if another thread executed the corresponds [CHECK](#) instruction.

```
void imply_thread_not_executed (uint k, uint t)
{
    Next next(k, t);
    next[State::full] = reset(&full, &flush, k, t);
    std::vector<std::string> stmts;
    for (uint pc = 0; pc < programs.size(); pc++)
        stmts.push_back(restore(&stmt, k, t, pc));
    next[State::stmt] = land(stmts);
    next.erase(next.find(State::heap));
    next.erase(next.find(State::exit));

    formula << assertion(imply(lnot(thread(k - 1, t)), next)) << '\n';
}
```

Example: `imply_thread_not_executed(1, 0)`

```
(assert (=> (not thread_0_0)
    (and (= accu_1_0 accu_0_0)
        (= mem_1_0 mem_0_0)
        (= adr_1_0 adr_0_0)
        (= val_1_0 val_0_0)
        (= full_1_0 (ite flush_0_0 false full_0_0))
        (and (= stmt_1_0_0 stmt_0_0_0)
            (= stmt_1_0_1 stmt_0_0_1)
            (= stmt_1_0_2 stmt_0_0_2)
            (= stmt_1_0_3 stmt_0_0_3)
            (= stmt_1_0_4 stmt_0_0_4))
        (= block_1_0_0 (ite check_0_0 false block_0_0_0))
        (= halt_1_0 halt_0_0))))
```


With all local thread states already being set, a store buffer flush just needs to write the heap array heap^k and disable the exit flag exit^k to continue execution.

```
void imply_thread_flushed (uint k, uint t)
{
    formula << assertion(imply(flush(k - 1, t),
                                land(equal(heap(k),
                                             store(heap(k - 1),
                                                  adr(k - 1, t),
                                                  val(k - 1, t)))
                                lnot(exit(k)))))) << '\n';
}
```

Example: `imply_thread_flushed(1, 0)`

```
(assert (=> flush_0_0
           (and (= heap_1 (store heap_0 adr_0_0 val_0_0))
                (not exit_1))))
```

Finally, to keep the model consistent if the machine terminated in a step $k < \text{bound}$, the heap array heap^{k-1} must be preserved. On the other hand, if the machine is still running in the final step, we need to set the default exit code as it would otherwise remain undefined.

```
void imply_machine_exited (uint k)
{
    formula << assertion(imply(exit(k - 1),
                                land(restore(&heap, k),
                                       exit(k)))) << '\n';

    if (k == bound)
        formula << assertion(imply(lnot(exit(k)),
                                    equal(exit_code, consth(0))))
            << '\n';
}
```

Example: `imply_machine_exited(1, 0)`

```
(assert (=> exit_0 (and (= heap_1 heap_0) exit_1)))
```

Example: `imply_machine_exited(17, 0)`

```
(assert (=> exit_16 (and (= heap_17 heap_16) exit_17)))
(assert (=> (not exit_17) (= exit-code #x0000)))
```

8 BTOR2

We also included the possibility to generate the word level model checking format BTOR2 [7], using the `-e btor2` command line parameter. It provides a *sequential extension* for specifying states in combination with their transition functions, which are automatically unrolled via symbolic substitution by the accompanying bounded model checker BtorMC. In contrast to the linear growth of our SMT-LIB encodings, the generated formulas size therefore remains constant for any given bound. We omit further details about the encoding process, as it is more or less identical to the previous using our functional next state logic, but a bit more tedious due to the line based syntax. Instead, we review the encoding of our store buffer litmus test example to highlight the pros and cons of BTOR2.

BTOR2 formulas consist of nodes, one per line, each prefixed with a strictly increasing numeric identifier serving as the input to subsequent nodes. By requiring every node to be defined before being used, the resulting direct acyclic graph is easy to parse and since every part of the expression may be referenced at a later point, no additional auxiliary variables are needed. See [7] for a detailed format description and the list of available nodes.

The encoding of our store buffer litmus test example given in Table 6 starts by defining the required boolean, bit-vector and array sorts.

```
1 sort bitvec 1  $\mathcal{B}^1$ 
2 sort bitvec 16  $\mathcal{B}^{16}$ 
3 sort array 2 2  $\mathcal{A}^{16}$ 
```

Next, a special set of input nodes is used to define the required boolean

```
4 zero 1
5 one 1
```

and bit-vector constants.

```
6 zero 2
7 one 2
8 constd 2 5
9 constd 2 17
```

In order to initialize our `heap` array with the contents of this example's memory map given in Listing 8, an uninitialized array state is declared and the corresponding elements written accordingly. By omitting its transition function, it is treated as a primary input only used during initialization and can safely be ignored in later steps.

```

10 state 3      mmap
11 write 3 10 6 6 mmap[0] ← 0
12 write 3 11 7 6 mmap[1] ← 0

```

Since our actual machine states are used in the definition of constraints and transition functions, they must be declared in advance.

- Accumulator registers:

```

13 state 2 accu0
14 state 2 accu1
15 state 2 accu2

```

- CAS memory registers:

```

16 state 2 mem0
17 state 2 mem1
18 state 2 mem2

```

- Store buffer address registers:

```

19 state 2 adr0
20 state 2 adr1
21 state 2 adr2

```

- Store buffer value registers:

```

22 state 2 val0
23 state 2 val1
24 state 2 val2

```

- Store buffer full flags:

```

25 state 1 full0
26 state 1 full1
27 state 1 full2

```

- Statement activation flags:

```

28 state 1 stmt0,0
29 state 1 stmt0,1
30 state 1 stmt0,2
31 state 1 stmt0,3
32 state 1 stmt0,4

```

```

33 state 1 stmt1,0
34 state 1 stmt1,1
35 state 1 stmt1,2
36 state 1 stmt1,3
37 state 1 stmt1,4

```

```

38 state 1 stmt2,0
39 state 1 stmt2,1
40 state 1 stmt2,2
41 state 1 stmt2,3
42 state 1 stmt2,4
43 state 1 stmt2,5

```

- Block flags:

```

44 state 1 block0,0
45 state 1 block0,1
46 state 1 block0,2

```

- Halt flags:

```

47 state 1 halt0
48 state 1 halt1
49 state 1 halt2

```

- Shared memory:

```

50 state 3 heap

```

- Exit flag:

```

51 state 1 exit

```

- Exit code:

```

52 state 2 exit-code

```

Same applies to the free transition variables thread_t and flush_t serving as the model's input.

```

53 input 1 thread0
54 input 1 thread1
55 input 1 thread2

```

```

56 input 1 flush0
57 input 1 flush1
58 input 1 flush2

```

In contrast to the SMT-LIB encoding, no additional declarations are needed for the introduction of helper variables and our execution variables $\text{exec}_{t,pc}$ therefore defined as simple **and** nodes.

```

59 and 1 28 53      stmt0,0 ∧ thread0 ≡ exec0,0
60 and 1 29 53      stmt0,1 ∧ thread0 ≡ exec0,1
61 and 1 30 53      stmt0,2 ∧ thread0 ≡ exec0,2
62 and 1 31 53      stmt0,3 ∧ thread0 ≡ exec0,3
63 and 1 32 53      stmt0,4 ∧ thread0 ≡ exec0,4

64 and 1 33 54      stmt1,0 ∧ thread1 ≡ exec1,0
65 and 1 34 54      stmt1,1 ∧ thread1 ≡ exec1,1
66 and 1 35 54      stmt1,2 ∧ thread1 ≡ exec1,2
67 and 1 36 54      stmt1,3 ∧ thread1 ≡ exec1,3
68 and 1 37 54      stmt1,4 ∧ thread1 ≡ exec1,4

69 and 1 38 55      stmt2,0 ∧ thread2 ≡ exec2,0
70 and 1 39 55      stmt2,1 ∧ thread2 ≡ exec2,1
71 and 1 40 55      stmt2,2 ∧ thread2 ≡ exec2,2
72 and 1 41 55      stmt2,3 ∧ thread2 ≡ exec2,3
73 and 1 42 55      stmt2,4 ∧ thread2 ≡ exec2,4
74 and 1 43 55      stmt2,5 ∧ thread2 ≡ exec2,5

```

The fixed arity of BTOR2 operators imposes a minor inconvenience if the same connective has to be applied over a larger number of variables. For example, a cascade of **and** nodes is required to build the conjunction over all block flags $\text{block}_{0,t}$ representing the checkpoint variable check_0 .

```

75 and 1 44 45      block0,0 ∧ block0,1
76 and 1 46 75      ⊢ ∧ block0,2 ≡ check0

```

Similarly, the disjunction serving as the *at-least-one* predicate of our *exactly-one* constraint is defined by a cascade of **or** nodes

```

77 or 1 53 54       thread0 ∨ thread1
78 or 1 55 77       ⊢ ∨ thread2
79 or 1 56 78       ⊢ ∨ flush0
80 or 1 57 79       ⊢ ∨ flush1
81 or 1 58 80       ⊢ ∨ flush2
82 or 1 51 81       ⊢ ∨ exit

```

and added as an invariant through the keyword **constraint**.

```

83 constraint 82

```

Definition of the remaining *at-most-one* predicate starts with the declaration of auxiliary input variables required by Carsten Sinz's sequential counter constraint $LT_{SEQ}^{7,1}$ [8],

```

84 input 1          thread0aux
85 input 1          thread1aux
86 input 1          thread2aux
87 input 1          flush0aux
88 input 1          flush1aux
89 input 1          flush2aux

```

followed by the corresponding set of clauses.

```

90 or 1 -53 84      ¬thread0 ∨ thread0aux
91 or 1 -54 85      ¬thread1 ∨ thread1aux
92 or 1 -84 85      ¬thread0aux ∨ thread1aux
93 or 1 -54 -84     ¬thread1 ∨ ¬thread0aux
94 or 1 -55 86      ¬thread2 ∨ thread2aux
95 or 1 -85 86      ¬thread1aux ∨ thread2aux
96 or 1 -55 -85     ¬thread2 ∨ ¬thread1aux
97 or 1 -56 87      ¬flush0 ∨ flush0aux
98 or 1 -86 87      ¬thread2aux ∨ flush0aux
99 or 1 -56 -86     ¬flush0 ∨ thread2aux
100 or 1 -57 88     ¬flush1 ∨ flush0aux
101 or 1 -87 88     ¬flush0aux ∨ flush1aux
102 or 1 -57 -87    ¬flush1 ∨ ¬flush0aux
103 or 1 -58 89     ¬flush2 ∨ flush1aux
104 or 1 -88 89     ¬flush1aux ∨ flush2aux
105 or 1 -58 -88    ¬flush2 ∨ ¬flush1aux
106 or 1 -51 -89    ¬exit ∨ ¬flush2aux

```

The *exactly-one* constraint can now be completed by building a conjunction over aforementioned clauses

```

107 and 1 90 91     (¬thread0 ∨ thread0aux) ∧ (¬thread1 ∨ thread1aux)
108 and 1 92 107    ⊢ ∧ (¬thread0aux ∨ thread1aux)
109 and 1 93 108    ⊢ ∧ (¬thread1 ∨ ¬thread0aux)
110 and 1 94 109    ⊢ ∧ (¬thread2 ∨ thread2aux)
111 and 1 95 110    ⊢ ∧ (¬thread1aux ∨ thread2aux)
112 and 1 96 111    ⊢ ∧ (¬thread2 ∨ ¬thread1aux)
113 and 1 97 112    ⊢ ∧ (¬flush0 ∨ flush0aux)
114 and 1 98 113    ⊢ ∧ (¬thread2aux ∨ flush0aux)
115 and 1 99 114    ⊢ ∧ (¬flush0 ∨ thread2aux)
116 and 1 100 115   ⊢ ∧ (¬flush1 ∨ flush0aux)

```

```

117 and 1 101 116       $\hookrightarrow \wedge (\neg \text{flush}_0^{\text{aux}} \vee \text{flush}_1^{\text{aux}})$ 
118 and 1 102 117       $\hookrightarrow \wedge (\neg \text{flush}_1 \vee \neg \text{flush}_0^{\text{aux}})$ 
119 and 1 103 118       $\hookrightarrow \wedge (\neg \text{flush}_2 \vee \text{flush}_1^{\text{aux}})$ 
120 and 1 104 119       $\hookrightarrow \wedge (\neg \text{flush}_1^{\text{aux}} \vee \text{flush}_2^{\text{aux}})$ 
121 and 1 105 120       $\hookrightarrow \wedge (\neg \text{flush}_2 \vee \neg \text{flush}_1^{\text{aux}})$ 
122 and 1 106 121       $\hookrightarrow \wedge (\neg \text{exit} \vee \neg \text{flush}_2^{\text{aux}})$ 

```

and adding yet another invariant.

```

123 constraint 122

```

This cardinality constraint is then again further influenced by prohibiting certain transitions like flushing an empty store buffer or executing barrier instructions if it is full,

```

124 or 1 29 32          stmt0,1  $\vee$  stmt0,4
125 implies 1 124 -53  $\hookrightarrow \implies \neg \text{thread}_0$ 
126 ite 1 25 125 -56 ite(full0,  $\hookrightarrow$ ,  $\neg \text{flush}_0$ )
127 constraint 126

```

```

128 or 1 34 37          stmt1,1  $\vee$  stmt1,4
129 implies 1 128 -54  $\hookrightarrow \implies \neg \text{thread}_1$ 
130 ite 1 26 129 -57 ite(full1,  $\hookrightarrow$ ,  $\neg \text{flush}_1$ )
131 constraint 130

```

```

132 implies 1 58 27     flush2  $\implies$  full2
133 constraint 132

```

executing a thread while it is waiting for a checkpoint,

```

134 and 1 44 -76        block0,0  $\wedge \neg \text{check}_0$ 
135 implies 1 134 -53  $\hookrightarrow \implies \neg \text{thread}_0$ 
136 constraint 135

```

```

137 and 1 45 -76        block0,1  $\wedge \neg \text{check}_0$ 
138 implies 1 137 -54  $\hookrightarrow \implies \neg \text{thread}_1$ 
139 constraint 138

```

```

140 and 1 46 -76        block0,2  $\wedge \neg \text{check}_0$ 
141 implies 1 140 -55  $\hookrightarrow \implies \neg \text{thread}_2$ 
142 constraint 141

```

and stopping halted threads from being scheduled.

```

143 implies 1 47 -53  halt0  $\implies$   $\neg$ thread0
144 constraint 143

145 implies 1 48 -54  halt1  $\implies$   $\neg$ thread1
146 constraint 145

147 implies 1 49 -55  halt2  $\implies$   $\neg$ thread2
148 constraint 147

```

As mentioned earlier, the main advantage of the BTOR2 format lies in the ability to define states in combination with their transition function. In our case, definition of machine states therefore follows a pretty basic pattern: after initializing the state with the keyword **init**, its successor is defined through the appropriate set of nodes and the resulting expression registered as the given state's transition function by using the corresponding keyword **next**.

- Accumulator registers:

```

149 init 2 13 6      accu00  $\leftarrow$  0
150 add 2 13 7      accu0 + 1
151 ite 2 59 150 13  ite(exec0,0,  $\downarrow$ , accu0)  $\equiv$  ADDI0
152 read 2 50 7     read(1)
153 eq 1 19 7       adr0 = 1
154 and 1 25 153     $\downarrow \wedge$  full0
155 ite 2 154 22 152 ite( $\downarrow$ , val0, read(1))
156 ite 2 61 155 151 ite(exec0,2,  $\downarrow$ , ADDI0)
157 next 2 13 156   accu0'  $\leftarrow$   $\downarrow$ 

158 init 2 14 6      accu10  $\leftarrow$  0
159 add 2 14 7      accu1 + 1
160 ite 2 64 159 14  ite(exec1,0,  $\downarrow$ , accu1)  $\equiv$  ADDI1
161 read 2 50 6     read(0)
162 eq 1 20 6       adr1 = 0
163 and 1 26 162     $\downarrow \wedge$  full1
164 ite 2 163 23 161 ite( $\downarrow$ , val1, read(0))
165 ite 2 66 164 160 ite(exec1,2,  $\downarrow$ , ADDI1)
166 next 2 14 165   accu1'  $\leftarrow$   $\downarrow$ 

167 init 2 15 6      accu20  $\leftarrow$  0
168 eq 1 21 6       adr2 = 0
169 and 1 27 168     $\downarrow \wedge$  full2
170 ite 2 169 24 161 ite( $\downarrow$ , val2, read(0))

```



```

171 add 2 15 170       $\downarrow + \text{accu}_2$ 
172 ite 2 70 171 15    $\text{ite}(\text{exec}_{2,1}, \downarrow, \text{accu}_2) \equiv \text{ADD}$ 
173 eq 1 21 7          $\text{adr}_2 = 1$ 
174 and 1 27 173       $\downarrow \wedge \text{full}_2$ 
175 ite 2 174 24 152   $\text{ite}(\downarrow, \text{val}_2, \text{read}(1))$ 
176 add 2 15 175       $\downarrow + \text{accu}_2$ 
177 ite 2 71 176 172   $\text{ite}(\text{exec}_{2,2}, \downarrow, \text{ADD})$ 
178 next 2 15 177      $\text{accu}'_2 \leftarrow \downarrow$ 

```

- CAS memory registers:

```

179 init 2 16 6        $\text{mem}_0^0 \leftarrow 0$ 
180 next 2 16 16       $\text{mem}'_0 \leftarrow \text{mem}_0$ 

181 init 2 17 6        $\text{mem}_1^0 \leftarrow 0$ 
182 next 2 17 17       $\text{mem}'_1 \leftarrow \text{mem}_1$ 

183 init 2 18 6        $\text{mem}_2^0 \leftarrow 0$ 
184 next 2 18 18       $\text{mem}'_2 \leftarrow \text{mem}_2$ 

```

- Store buffer address registers:

```

185 init 2 19 6        $\text{adr}_0^0 \leftarrow 0$ 
186 ite 2 60 6 19       $\text{ite}(\text{exec}_{0,1}, 0, \text{adr}_0)$ 
187 next 2 19 186      $\text{adr}'_0 \leftarrow \downarrow$ 

188 init 2 20 6        $\text{adr}_1^0 \leftarrow 0$ 
189 ite 2 65 7 20       $\text{ite}(\text{exec}_{1,1}, 0, \text{adr}_1)$ 
190 next 2 20 189      $\text{adr}'_1 \leftarrow \downarrow$ 

191 init 2 21 6        $\text{adr}_2^0 \leftarrow 0$ 
192 next 2 21 21       $\text{adr}'_2 \leftarrow \text{adr}_2$ 

```

- Store buffer value registers:

```

193 init 2 22 6        $\text{val}_0^0 \leftarrow 0$ 
194 ite 2 60 13 22      $\text{ite}(\text{exec}_{0,1}, \text{accu}_0, \text{val}_0)$ 
195 next 2 22 194      $\text{val}'_0 \leftarrow \downarrow$ 

196 init 2 23 6        $\text{val}_1^0 \leftarrow 0$ 
197 ite 2 65 14 23      $\text{ite}(\text{exec}_{1,1}, \text{accu}_1, \text{val}_1)$ 
198 next 2 23 197      $\text{val}'_1 \leftarrow \downarrow$ 

199 init 2 24 6        $\text{val}_2^0 \leftarrow 0$ 
200 next 2 24 24       $\text{val}'_2 \leftarrow \text{val}_2$ 

```

- Store buffer full flags:

```

201 init 1 25 4      full00 ← 0
202 or 1 60 25      exec0,1 ∨ full0
203 ite 1 56 4 202   ite(flush0, 0, ↱)
204 next 1 25 203    full0' ← ↱

205 init 1 26 4      full10 ← 0
206 or 1 65 26      exec1,1 ∨ full1
207 ite 1 57 4 206   ite(flush1, 0, ↱)
208 next 1 26 207    full1' ← ↱

209 init 1 27 4      full20 ← 0
210 ite 1 58 4 27     ite(flush2, 0, full2)
211 next 1 27 210    full2' ← ↱

```

- Statement activation flags:

```

212 init 1 28 5      stmt0,00 ← 1
213 and 1 28 -59     stmt0,0 ∧ ¬exec0,0
214 next 1 28 213    stmt0,0' ← ↱

215 init 1 29 4      stmt0,10 ← 0
216 and 1 29 -60     stmt0,1 ∧ ¬exec0,1
217 ite 1 28 59 216   ite(stmt0,0, exec0,0, ↱)
218 next 1 29 217    stmt0,1' ← ↱

219 init 1 30 4      stmt0,20 ← 0
220 and 1 30 -61     stmt0,2 ∧ ¬exec0,2
221 ite 1 29 60 220   ite(stmt0,1, exec0,1, ↱)
222 next 1 30 221    stmt0,2' ← ↱

223 init 1 31 4      stmt0,30 ← 0
224 and 1 31 -62     stmt0,3 ∧ ¬exec0,3
225 ite 1 30 61 224   ite(stmt0,2, exec0,2, ↱)
226 next 1 31 225    stmt0,3' ← ↱

227 init 1 32 4      stmt0,40 ← 0
228 and 1 32 -63     stmt0,4 ∧ ¬exec0,4
229 ite 1 31 62       ite(stmt0,3, exec0,3, ↱)
230 next 1 32 229    stmt0,4' ← ↱

231 init 1 33 5      stmt1,00 ← 1

```

232	and	1	33	-64	$\text{stmt}_{1,0} \wedge \neg \text{exec}_{1,0}$
233	next	1	33	232	$\text{stmt}'_{1,0} \leftarrow \downarrow$
234	init	1	34	4	$\text{stmt}_{1,1}^0 \leftarrow 0$
235	and	1	34	-65	$\text{stmt}_{1,1} \wedge \neg \text{exec}_{1,1}$
236	ite	1	33	64	$\text{ite}(\text{stmt}_{1,0}, \text{exec}_{1,0}, \downarrow)$
237	next	1	34	236	$\text{stmt}'_{1,1} \leftarrow \downarrow$
238	init	1	35	4	$\text{stmt}_{1,2}^0 \leftarrow 0$
239	and	1	35	-66	$\text{stmt}_{1,2} \wedge \neg \text{exec}_{1,2}$
240	ite	1	34	65	$\text{ite}(\text{stmt}_{1,1}, \text{exec}_{1,1}, \downarrow)$
241	next	1	35	240	$\text{stmt}'_{1,2} \leftarrow \downarrow$
242	init	1	36	4	$\text{stmt}_{1,3}^0 \leftarrow 0$
243	and	1	36	-67	$\text{stmt}_{1,3} \wedge \neg \text{exec}_{1,3}$
244	ite	1	35	66	$\text{ite}(\text{stmt}_{1,2}, \text{exec}_{1,2}, \downarrow)$
245	next	1	36	244	$\text{stmt}'_{1,3} \leftarrow \downarrow$
246	init	1	37	4	$\text{stmt}_{1,4}^0 \leftarrow 0$
247	and	1	37	-68	$\text{stmt}_{1,4} \wedge \neg \text{exec}_{0,4}$
248	ite	1	36	67	$\text{ite}(\text{stmt}_{1,3}, \text{exec}_{1,3}, \downarrow)$
249	next	1	37	248	$\text{stmt}'_{1,4} \leftarrow \downarrow$
250	init	1	38	5	$\text{stmt}_{2,0}^0 \leftarrow 1$
251	and	1	38	-69	$\text{stmt}_{2,0} \wedge \neg \text{exec}_{2,0}$
252	next	1	38	251	$\text{stmt}'_{2,0} \leftarrow \downarrow$
253	init	1	39	4	$\text{stmt}_{2,1}^0 \leftarrow 0$
254	and	1	39	-70	$\text{stmt}_{2,1} \wedge \neg \text{exec}_{2,1}$
255	ite	1	38	69	$\text{ite}(\text{stmt}_{2,0}, \text{exec}_{2,0}, \downarrow)$
256	next	1	39	255	$\text{stmt}'_{2,1} \leftarrow \downarrow$
257	init	1	40	4	$\text{stmt}_{2,2}^0 \leftarrow 0$
258	and	1	40	-71	$\text{stmt}_{2,2} \wedge \neg \text{exec}_{2,2}$
259	ite	1	39	70	$\text{ite}(\text{stmt}_{2,1}, \text{exec}_{2,1}, \downarrow)$
260	next	1	40	259	$\text{stmt}'_{2,2} \leftarrow \downarrow$
261	init	1	41	4	$\text{stmt}_{2,3}^0 \leftarrow 0$
262	and	1	41	-72	$\text{stmt}_{2,3} \wedge \neg \text{exec}_{2,3}$
263	ite	1	40	71	$\text{ite}(\text{stmt}_{2,2}, \text{exec}_{2,2}, \downarrow)$
264	next	1	41	263	$\text{stmt}'_{2,3} \leftarrow \downarrow$

265	init	1	42	4	$\text{stmt}_{2,4}^0 \leftarrow 0$
266	and	1	42	-73	$\text{stmt}_{2,4} \wedge \neg \text{exec}_{2,4}$
267	eq	1	15	6	$\text{accu}_2 = 0$
268	and	1	72	-267	$\text{exec}_{2,3} \wedge \text{accu}_2 \neq 0$
269	ite	1	41	268 266	$\text{ite}(\text{stmt}_{2,3}, \downarrow, \text{stmt}_{2,4} \wedge \neg \text{exec}_{2,4})$
270	next	1	42	269	$\text{stmt}_{2,4}' \leftarrow \downarrow$

271	init	1	43	4	$\text{stmt}_{2,5}^0 \leftarrow 0$
272	and	1	43	-74	$\text{stmt}_{2,5} \wedge \neg \text{exec}_{2,5}$
273	and	1	72	267	$\text{exec}_{2,3} \wedge \text{accu}_2 = 0$
274	ite	1	41	273 272	$\text{ite}(\text{stmt}_{2,3}, \downarrow, \text{stmt}_{2,5} \wedge \neg \text{exec}_{2,5})$
275	next	1	43	274	$\text{stmt}_{2,5}' \leftarrow \downarrow$

- Block flags:

276	init	1	44	4	$\text{block}_{0,0}^0 \leftarrow 0$
277	or	1	62	44	$\text{exec}_{0,3} \vee \text{block}_{0,0}$
278	ite	1	76	4 277	$\text{ite}(\text{check}_0, 0, \downarrow)$
279	next	1	44	278	$\text{block}_{0,0}' \leftarrow \downarrow$

280	init	1	45	4	$\text{block}_{0,1}^0 \leftarrow 0$
281	or	1	67	45	$\text{exec}_{1,3} \vee \text{block}_{0,1}$
282	ite	1	76	4 281	$\text{ite}(\text{check}_0, 0, \downarrow)$
283	next	1	45	282	$\text{block}_{0,1}' \leftarrow \downarrow$

284	init	1	46	4	$\text{block}_{0,2}^0 \leftarrow 0$
285	or	1	69	46	$\text{exec}_{2,0} \vee \text{block}_{0,2}$
286	ite	1	76	4 285	$\text{ite}(\text{check}_0, 0, \downarrow)$
287	next	1	46	286	$\text{block}_{0,2}' \leftarrow \downarrow$

- Halt flags:

288	init	1	47	4	$\text{halt}_0^0 \leftarrow 0$
289	or	1	63	47	$\text{exec}_{0,4} \vee \text{halt}_0$
290	next	1	47	289	$\text{halt}_0' \leftarrow \downarrow$

291	init	1	48	4	$\text{halt}_1^0 \leftarrow 0$
292	or	1	68	48	$\text{exec}_{1,4} \vee \text{halt}_1$
293	next	1	48	292	$\text{halt}_1' \leftarrow \downarrow$

294	init	1	49	4	$\text{halt}_2^0 \leftarrow 0$
295	next	1	49	49	$\text{halt}_2' \leftarrow \text{halt}_2$

- Shared memory:

```

296 init 3 50 12      heap0 ← mmap
297 write 3 50 19 22  write(adr0, val0)
298 ite 3 56 297 50    ite(flush0, ⊥, heap) ≡ FLUSH0
299 write 3 50 20 23  write(adr1, val1)
300 ite 3 57 299 298    ite(flush1, ⊥, FLUSH0) ≡ FLUSH1
301 write 3 50 21 24  write(adr2, val2)
302 ite 3 58 301 300    ite(flush2, ⊥, FLUSH1)
303 next 3 50 302      heap' ← ⊥

```

- Exit flag:

```

304 init 1 51 4      exit0 ← 0
305 and 1 289 292     halt'0 ∧ halt'1
306 and 1 49 305      ⊥ ∧ halt2
307 or 1 51 306       ⊥ ∨ exit
308 or 1 73 307       ⊥ ∨ exec2,4
309 or 1 74 308       ⊥ ∨ exec2,5
310 next 1 51 309     exit' ← ⊥

```

- Exit code:

```

311 init 2 52 6      exit-code0 ← 0
312 ite 2 73 6 52     ite(exec2,4, 0, exit-code)
313 ite 2 74 7 312     ite(exec2,5, 1, ⊥)
314 next 2 52 313     exit-code' ← ⊥

```

Finally, we declare an exit code greater than zero as the only **bad** state.

```

315 neq 1 6 52
316 bad 315

```

9 Testing and Debugging

The basic functionality of our toolchain is validated by a total of 800 test cases. While fixing C++17 related errors was relatively trivial due to the broad range of available tools, debugging the generated SMT encodings is a bit more tedious since cause and effect have to be determined manually by inspecting rather comprehensive models and formulas.

In order to automatically validate the correctness of our encodings, we included a `replay` mode for reevaluating the execution sequence of a given trace by the virtual machine used for simulation. If a mismatch is found, it returns the first step in which the results start to differ and prints the corresponding states of both execution traces for comparison. This mode also simplified the encodings' development significantly, since the condensed error trace helps to narrow down the potentially flawed SMT expressions.

Of course, only satisfiable formulas may be replayed. If the outcome of a valid execution sequence is accidentally unsatisfiable and therefore yields no trace, we still have to manually inspect the formula. Unfortunately, no SMT solver supported the extraction of an *unsatisfiable core* at the time we developed our encodings, but this feature starts to get implemented in recent solvers like Bitwuzla [9]. However, without any additional constraints, the only reason our formulas can get unsatisfiable is due to a violation of the cardinality constraint and investigations may therefore be concentrated on expressions containing transition variables.

Another frequent cause for unintended results lies in the nature of bounded model checking. Before expecting an error in the encoding, it is highly advisable to make sure that the targeted states are indeed reachable by the chosen bound. For example, when a model for the `CAS` variant of our concurrent counter experiments (described in the next section) was found during our initial trials, we immediately started to panic and questioned our encoding. Since such situations already commonly occurred during development, we remembered the following proverb which emerged during that time and soon realized that the problem indeed was yet another insufficient bound.

If you ask yourself – “How can this be?” – maybe just the bound's too wee!

10 Experiments

To asses performance related aspects of our encodings, we conducted a series of experiments, using the following versions of supported SMT solvers.

- Boolector 3.2.1³ (including BtorMC)
- CVC4 1.8⁴
- Z3 4.8.9⁵

We recorded the resulting formula sizes in terms of the number of generated expression, as well as the runtimes for encoding and solving each particular instance. All tests were performed on a cluster of Intel Xeon[®] E5-2620 v4 nodes, with CPU runtime and memory usage limited to 86400 seconds (24 hours) and 8 GB respectively.

Litmus Tests

Instead of a rigorous formal description, the memory ordering principles of Intel’s [3] and AMD’s [4] x86 implementations are defined by example through a set of so called “litmus tests”. In order to show conformance with the memory ordering model of our virtual machine, the test suites have been ported to ConcuBinE and details are available in Appendices A and B.

№	bound	btor2		functional		relational	
		time	size	time	size	time	size
1	9	0.01	166	6.38	2051	10.18	2404
2	10	0.69	176	7.04	2508	9.51	2950
3	10	0.75	174	6.61	2408	7.79	2850
4	5	0.32	93	1.95	627	2.41	724
5	12	0.98	194	9.62	3228	12.71	3950
6	13	0.76	256	12.62	4889	16.51	5723
7	14	0.94	322	16.15	6770	21.15	7780
8	12	0.93	334	12.83	6000	18.24	6890
9	8	0.01	186	4.71	2058	7.09	2428
10	8	0.28	173	3.40	1938	5.72	2252

Table 8: Intel litmus test encoding times in milliseconds and formula sizes in terms of the number of expressions.

³<https://github.com/Boolector/boolector>

⁴<https://github.com/CVC4/CVC4>

⁵<https://github.com/Z3Prover/z3>

Bounds and encoder statistics of both test suites are given in Tables 8 and 9. Comparing the sizes of generated formulas, visualized in Figures 8 and 9, clearly shows the main advantage of using BTOR2: compact problem instances and reduced computational complexity of its encoding process due to being automatically unrolled for any given bound via symbolic substitution by BtorMC at runtime. Furthermore, the additional redundancy introduced by the relational next state logic causes a quite substantial gap between the size of our SMT-LIB encoding variants.

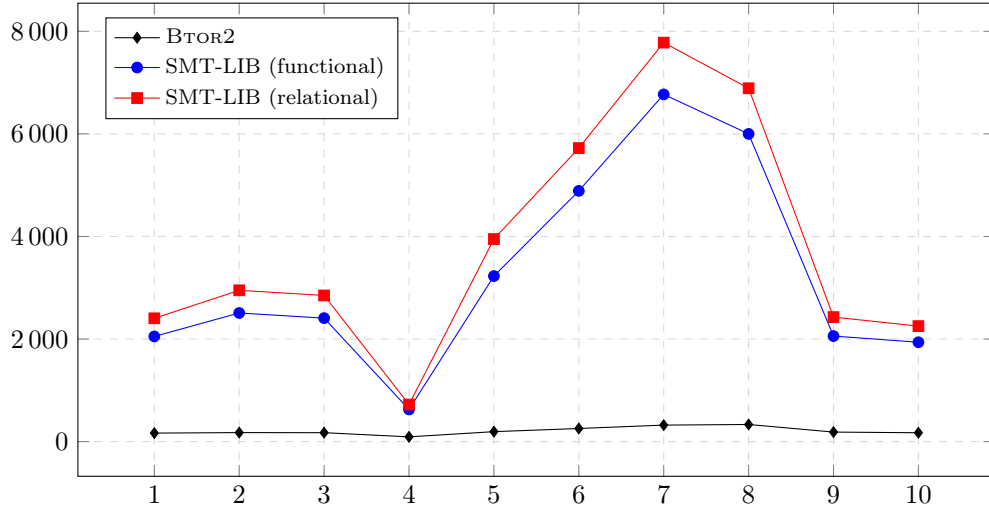


Figure 8: Intel litmus test formula sizes in terms of the number of expressions.

№	bound	btor2		functional		relational	
		time	size	time	size	time	size
1	9	0.05	166	4.79	2051	6.99	2404
2	10	0.06	176	6.76	2508	8.73	2950
3	16	0.01	206	12.54	4536	18.64	5818
4	10	0.01	174	7.96	2408	8.75	2850
5	12	0.05	188	8.93	3108	11.94	3830
6	14	1.47	322	16.36	6770	21.21	7780
7	13	0.01	256	12.39	4889	14.74	5723
8	12	0.61	196	9.07	3348	12.10	4070
9	14	0.66	210	11.59	4168	16.63	5290

Table 9: AMD litmus test encoding times in milliseconds and formula sizes in terms of the number of expressions.

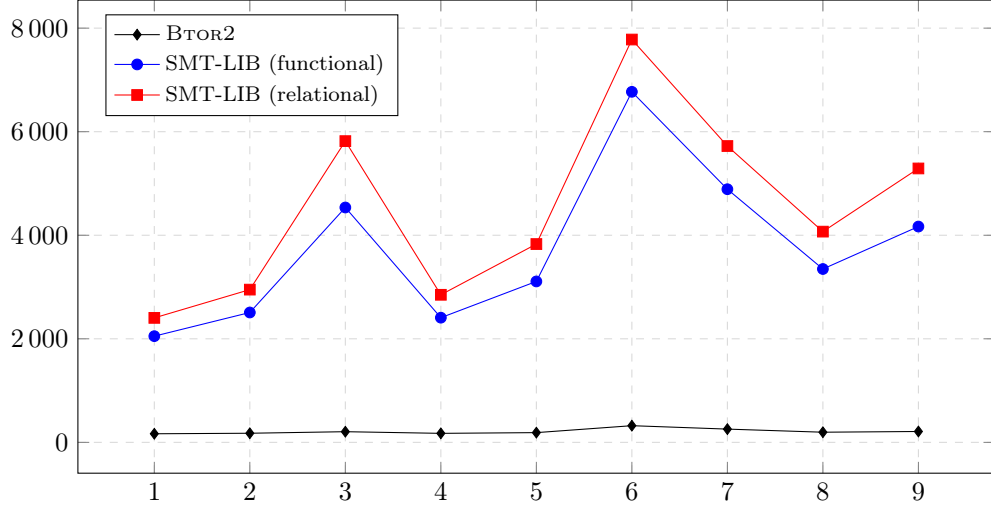


Figure 9: AMD litmus test formula sizes in terms of the number of expressions.

All litmus tests passed and the claim, that the encodings of our virtual machine model conform with the memory ordering principles of both major x86 vendors therefore validated by example. The corresponding solving times, given in Tables 10 and 11, reveal a dramatic difference between the tested solvers and encoding variants, even forcing us to resort to a logarithmic scale for plotting runtimes as shown in Figures 10 and 11 to get a meaningful graphical representation. Our functional encoding (BTOR2 and SMT-LIB) turned out to be the fastest, with BtorMC on top, closely followed by Boolector and Z3. As expected, the relational SMT-LIB approach is somewhat slower, but still beats CVC4 for any encoding variant when using Boolector or Z3. After reporting this observation to members of CVC4’s development team it was confirmed that the latest release still uses a customized version of MiniSAT⁶ as SAT backend for QF_AUFBV formulas, which seems to be the main reason for the performance gap in comparison to Boolector (incorporating CaDiCaL⁷ 1.0.3) and Z3.

⁶<https://github.com/niklasso/minisat>

⁷<https://github.com/arminbiere/cadical>

	BtorMC	Boolector		Z3		CVC4	
Nº	btor2	functional	relational	functional	relational	functional	relational
1	0.01	0.01	0.06	0.02	0.15	0.05	0.22
2	0.01	0.02	0.05	0.03	0.29	0.12	0.27
3	0.02	0.05	1.79	0.04	0.20	0.80	1.29
4	0.01	0.01	0.01	0.02	0.03	0.02	0.03
5	0.05	0.08	2.54	0.11	0.38	4.87	2.22
6	0.03	0.05	0.68	0.07	1.13	0.63	2.91
7	0.11	0.16	3.86	0.45	8.10	12225.10	35213.50
8	0.05	0.07	10.80	0.18	4.49	17415.50	26895.40
9	0.01	0.03	0.64	0.03	0.25	0.36	12.42
10	0.01	0.01	0.15	0.03	0.13	0.15	0.22

Table 10: Intel litmus test solving times in seconds.

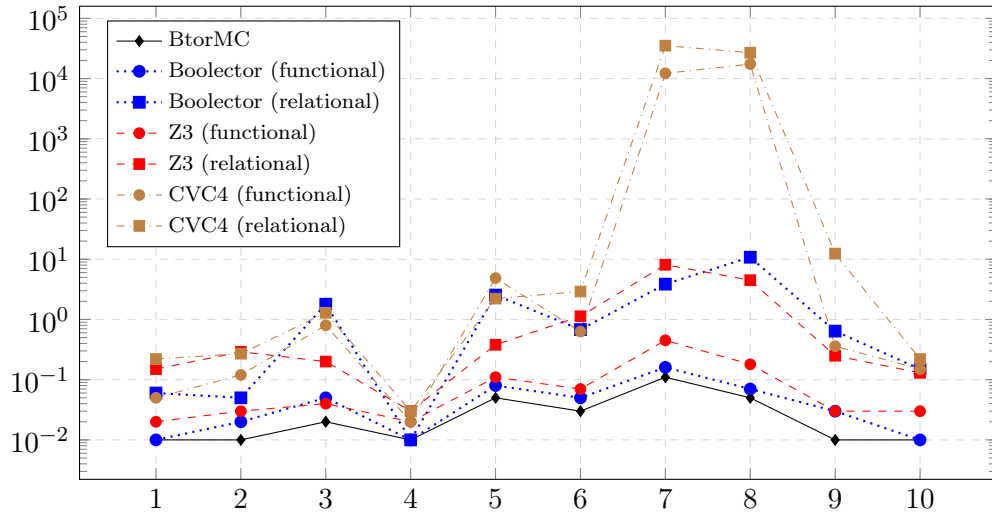


Figure 10: Intel litmus test solving times in seconds.

	BtorMC	Boolector		Z3		CVC4	
Nº	btor2	functional	relational	functional	relational	functional	relational
1	0.01	0.01	0.03	0.01	0.14	0.05	0.22
2	0.01	0.03	0.06	0.03	0.26	0.11	0.27
3	0.08	0.10	5.71	0.09	1.08	1.82	4.67
4	0.01	0.02	2.64	0.04	0.22	0.21	0.32
5	0.05	0.06	1.58	0.15	0.90	1.90	127.06
6	0.10	0.16	3.86	0.45	8.12	12213.70	35524.30
7	0.05	0.07	0.71	0.09	1.05	0.58	2.84
8	0.04	0.10	3.03	0.07	0.35	2.46	4.06
9	0.10	0.10	2.11	0.46	1.16	4.28	3064.73

Table 11: AMD litmus test solving times in seconds.

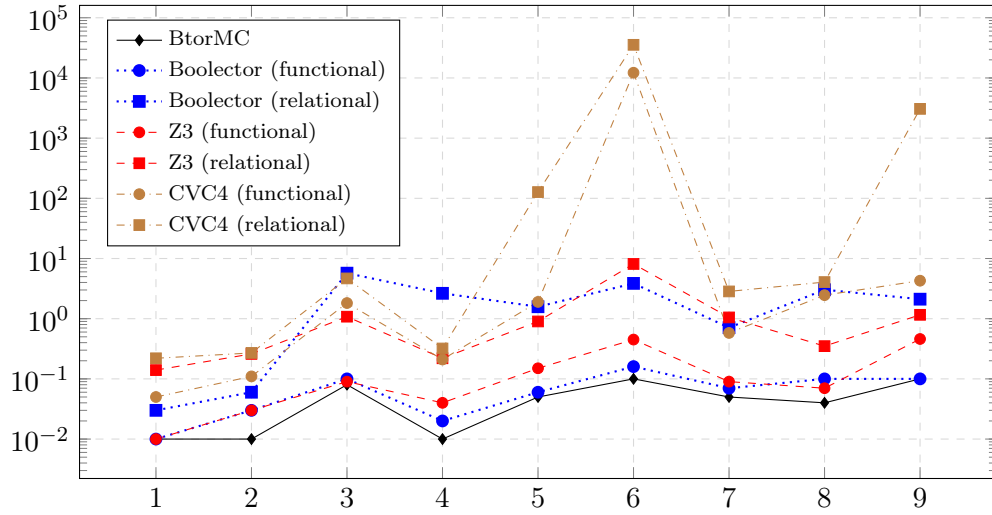



Figure 11: AMD litmus test solving times in seconds.

Concurrent Counter

To show the scalability of our approach, we also use a parametrized concurrent counter (inspired by Paul McKenney [2]), where m threads increment a shared variable n times. Two flavours were tested: a faulty version using **STORE** to compare runtimes of satisfiable instances and a valid one relying on **CAS**, resulting in unsatisfiable runs. Both were executed for any combination of $2 \leq m \leq 4$ threads and $2 \leq n \leq 4$ increments respectively, forming a set of parameter configurations large enough to yield representative results while still being solvable within the given time and memory limits.

```
inc: LOAD 0
      ADDI 1
      STORE 0
      LOAD <adr>
      SUBI 1
      STORE <adr>
      JNZ inc
      CHECK 0
      HALT
```



Listing 10: Faulty Counter Template

Listing 10 shows the faulty counter template. Each thread t starts by loading, incrementing and storing the shared counter at address 0. Next, the local counter (limiting the number of iterations) is loaded, decremented and simply stored at the address specified by the template parameter **<adr>**, which is replaced by $t+10$ for each thread $0 \leq t < m$. If this local counter (initialized with n) remains greater than zero, the counting process is restarted by jumping back to the initial program statement. Otherwise, the threads synchronize upon checkpoint 0, signaling that counting has finished and the result should be evaluated. The checker thread given in Listing 11 then compares the shared counter at address 0 to the expected value $m * n$ supplied by yet another template parameter **<sum>** and exits 1 if they differed.

```
CHECK 0
ADDI <sum>
CMP 0
JNZ error
EXIT 0
error: EXIT 1
```

Listing 11: Checker Template

In this example, relying on `STORE` to write a shared variable leads to an obvious data-race, even without the additional inconsistency introduced by the store buffer. Since an exit code greater than zero will serve as the bad state in the resulting model checking problem, we now must choose a proper bound such that the program will be fully executed and the checker thread’s exit statements can be reached. We therefore have to find the longest possible trace through the program, its worst-case execution time (WCET) so to say. For our faulty counter, the process is straightforward and it can simply be determined by adding n times the number of steps required by each iteration of the counting loop `inc` (7 instructions + 2 flushes) to the 2 instructions left for completing the counter thread’s execution. The minimum required bound can now be computed by multiplying the result with the number of participating threads m and adding the 5 steps needed by the checker thread, leading to the following equation.

$$m * (9 * n + 2) + 5$$

For example, $m = n = 2$ would require a bound of $2 * (9 * 2 + 2) + 5 = 45$.

m	n	bound	btor2		functional		relational	
			time	size	time	size	time	size
2	2	45	1.26	396	54.26	24783	80.01	33288
2	3	63	0.60	396	112.04	34539	111.30	46446
2	4	81	0.01	396	97.39	44295	146.15	59604
3	2	65	0.83	547	105.60	50284	166.88	67704
3	3	92	1.62	547	187.94	70939	228.75	95595
3	4	119	1.80	546	131.82	91594	188.94	123486
4	2	85	1.92	664	111.56	76363	171.92	105858
4	3	121	0.01	663	160.77	108439	277.84	150426
4	4	157	2.07	664	208.47	140515	313.55	194994

Table 12: Faulty counter encoding times in milliseconds and formula sizes in terms of the number of expressions for increasing values of m and n .

Table 12 shows the bounds and encoder statistics of our faulty counter experiments. While encoder runtimes are rather negligible, Figure 12 again highlights the dramatic difference in size between BTOR2 and SMT-LIB encoding variants for increasing values of m and n . As expected, the number of participating threads m is the main driving factor for increasing the problem size. The off-by-one differences in the size of the BTOR2 encodings for an equal number of threads m can be explained by the reuse of constants necessary to specify local counter addresses and the expected final value.

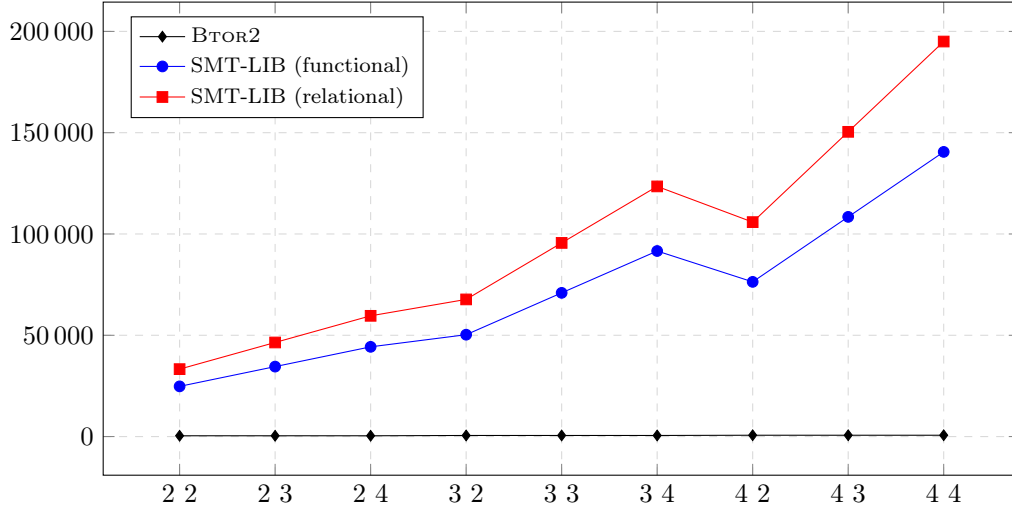


Figure 12: Faulty counter formula sizes in terms of the number of expressions for increasing values of m and n .

The time it took the supported solvers to find faulty traces for our faulty counter experiments are listed in Table 13 and visualized in Figure 13. Only BtorMC, Boolector using the functional and Z3 using the relational next state logic were able to find a solution for all experiments. Surprisingly, Z3 in combination with the functional next state logic performed much worse than expected and was just able to solve as many instances as Boolector using the relational variant. CVC4 however could not even solve a single instance within the given time and memory limits, again most likely due to the SAT solver used.

m n	BtorMC	Boolector		Z3		CVC4	
	btor2	functional	relational	functional	relational	functional	relational
2 2	0.51	0.82	1838.45	82.94	19.76	-	-
2 3	8.41	10.69	2173.17	460.48	54.97	-	-
2 4	21.61	29.18	2367.87	1775.55	90.35	-	-
3 2	15.73	23.73	13990.70	4527.69	277.55	-	-
3 3	80.74	82.19	66764.00	13899.70	781.81	-	-
3 4	171.25	165.11	-	-	3308.92	-	-
4 2	157.16	158.09	-	-	1555.43	-	-
4 3	434.82	493.33	-	-	7060.05	-	-
4 4	1023.00	1037.32	-	-	35431.10	-	-

Table 13: Faulty counter solving times in seconds for increasing values of m and n .

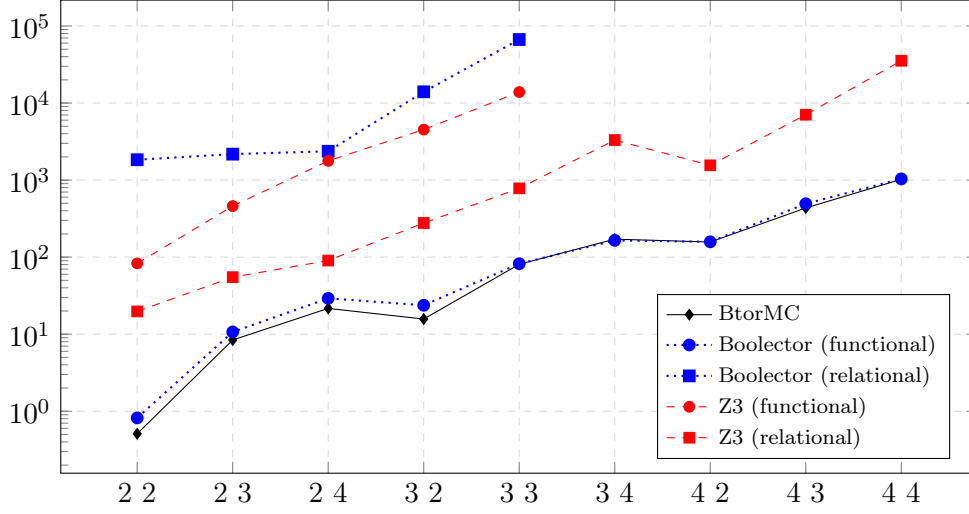


Figure 13: Faulty counter solving times in seconds for increasing values of m and n .

Finally, we also experimented with a valid version of the previous concurrent counter example, given in Listing 12, which resorts to *compare-and-swap* for writing a shared variable in a predictable manner. Instead of simply reading the shared counter's value, we now use `MEM` to additionally memorize it for the latter application of `CAS`. Since `CAS` might fail due to an intermediate alteration of the shared counter variable by another thread, we must prevent an erroneous subsequent decrement of the thread's local counter variable by embedding it in a nested loop (labelled `cas`) and repeat the increment until it succeeds. Everything else remains the same, including the checker thread.

```

inc: MEM 0
    ADDI 1
    CAS 0
    JZ inc
    LOAD <adr>
    SUBI 1
    STORE <adr>
    JNZ inc
    CHECK 0
    HALT

```

$\left. \begin{array}{l} \text{cas} \\ \text{inc} \end{array} \right\}$

Listing 12: Valid Counter Template

Determining the required bound is a bit more complex, since we also have to account for the maximum number of failed **CAS** instructions to ensure that all possible traces are included in the search space. This worst case can occur if the participating counter threads are scheduled one after the other,

$$\text{thread}_0^0 \rightarrow \dots \rightarrow \text{thread}_{m-1}^{m-1} \rightarrow \text{thread}_0^m \rightarrow \dots \rightarrow \text{thread}_{m-1}^{2m-1} \rightarrow \dots$$

causing all but one (the first) to fail in each iteration of the counting loop **inc**. Consider the following example trace for $m = 2$ and $n = 1$, where counting to $m * n = 2$ may require up to three **cas** loop iterations in total.

Thread	CAS Status	Local Counter
0	done	0
1	failed	1
1	done	0

If we now examine the influence by an increased number of threads ($m = 3$ and $n = 1$), one notices that the maximum number of **cas** iterations follows a *triangular number*: $m * \frac{m+1}{2} = 6$.

Thread	CAS Status	Local Counter
0	done	0
1	failed	1
2	failed	1
1	done	1
2	failed	0
2	done	1

Since increasing the local increments n directly influences the number of **cas** iterations, we just need to multiply the previously identified *triangular number* by n to get the maximum: $n * m * \frac{m+1}{2}$. If we now consider the number of steps required by each iteration of the **cas** loop (4), the ones left to complete the counting loop **inc** (5) and the 2 instructions remaining in the counter thread's program, the final bound can be computed by the following equation.

$$m * (4 * n * m * \frac{m+1}{2} + 5 * n + 2) + 5$$

For example, the valid counter experiment for $m = n = 2$ already requires a bound of $(4 * 2 * 2 * \frac{2+1}{2} + 5 * 2 + 2) * 2 + 5 = 77$.

Bounds and encoder statistics for our valid counter experiments are given in Table 14 and formula sizes visualized in Figure 14. While the addition of a single instruction only slightly increases the formula sizes of our BTOR2 encoding, the SMT-LIB variants exhibit a significant blowup due to the larger bound required by the introduction of a nested loop. Naturally, encoder runtimes are also affected, but remain acceptable as all experiments could be encoded in less than a second.

m	n	bound	btor2		functional		relational	
			time	size	time	size	time	size
2	2	77	0.01	424	70.08	46453	98.68	63470
2	3	111	1.33	424	96.43	66785	67.35	91316
2	4	145	0.16	424	120.35	87117	87.92	119162
3	2	185	0.51	589	101.87	157645	156.29	216105
3	3	272	0.51	589	146.43	231508	231.16	317460
3	4	359	0.51	588	189.60	305371	304.00	418815
4	2	373	0.60	720	256.03	374775	391.62	528078
4	3	553	0.61	719	369.42	555315	607.02	782598
4	4	733	0.62	720	498.63	735855	774.74	1037118

Table 14: Valid counter encoding times in milliseconds and formula sizes in terms of the number of expressions for increasing values of m and n .

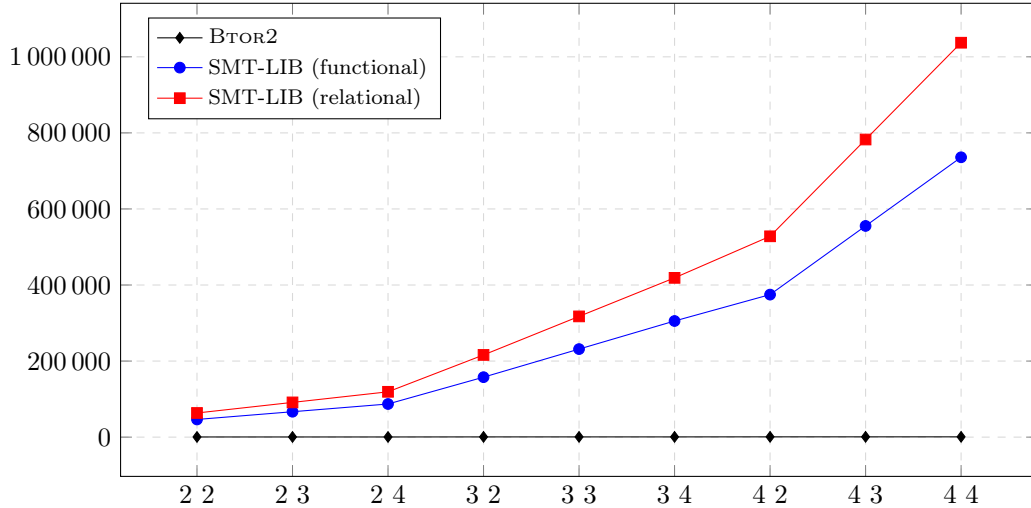


Figure 14: Valid counter formula sizes in terms of the number of expressions for increasing values of m and n .

Solving times of this experiment’s unsatisfiable runs are shown in Table 15 and visualized in Figure 15. Unfortunately, only 3 out of 9 experiments could be solved by BtorMC and Boolector within the given time and memory limits. This puts the practicability of our approach in question, since further increasing the program size, number of threads and bound required by even the tiniest real world example would generate problem instances which most likely won’t be solvable in a reasonable amount of time, possibly due to symmetric execution parts. Thus, these examples probably meet some kind of symmetry reduction or use a form of partial order reduction.

m n	BtorMC	Boolector		Z3		CVC4	
	btor2	functional	relational	functional	relational	functional	relational
2 2	2105.44	893.10	15283.50	-	-	-	-
2 3	10188.30	5706.20	-	-	-	-	-
2 4	56972.70	25617.10	-	-	-	-	-
3 2	-	-	-	-	-	-	-
3 3	-	-	-	-	-	-	-
3 4	-	-	-	-	-	-	-
4 2	-	-	-	-	-	-	-
4 3	-	-	-	-	-	-	-
4 4	-	-	-	-	-	-	-

Table 15: Valid counter solving times in seconds for increasing values of m and n .

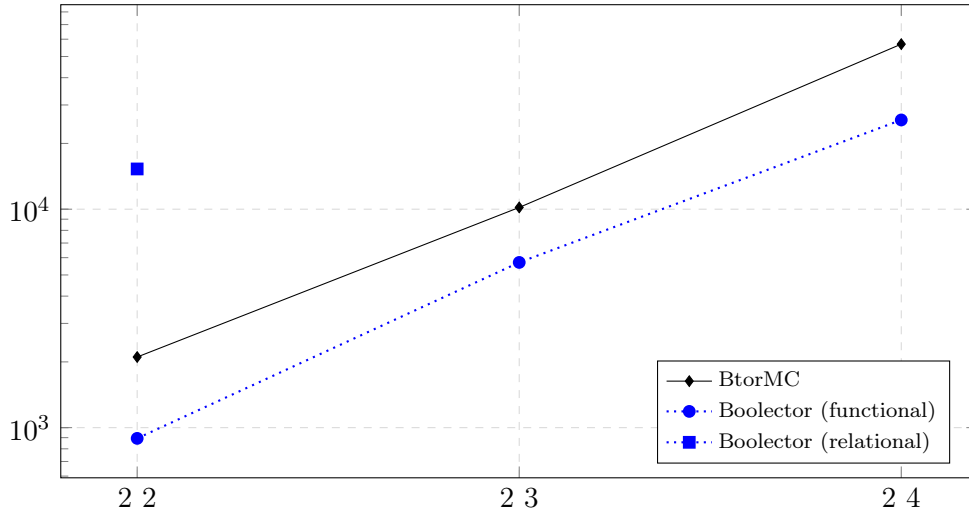


Figure 15: Valid counter solving times in seconds for increasing values of m and n .

11 Conclusion

In this thesis we explored the potential of bounded model checking for verifying concurrent programs including the target architecture’s memory ordering habits. Due to the complexity of actual processors, we devised a highly simplified virtual machine model and showed that the addition of store buffers is sufficient to imitate the behaviour of current x86 implementations.

After designing a formal framework for the execution of arbitrary programs on this virtual machine model, encodings of the corresponding model checking problems in two different SMT formats were developed. We then implemented ConcuBinE, a tool for automating the encoding and evaluation of generated SMT formulas using state-of-the-art solvers.

Even though experiments show the basic feasibility of our approach, the huge runtimes for even small examples puts its practicability in question. However, we still see it as an opportunity to integrate a similar checking procedure into compiler toolchains, automatically verifying the binaries of critical software with regards to the target architecture’s memory ordering model and other specifics.

Future work could therefore involve optimizing the encoding, devising a method for specifying assertions, developing a procedure to additionally compute the WCET for automatically determining the upper bound and applying the presented principles to a real architecture like Intel’s or AMD’s x86 processors.

References

- [1] Leslie Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, 1997.
- [2] Paul E. McKenney. Is Parallel Programming Hard, And, If So, What Can You Do About It? (v2017.01.02a). *CoRR*, abs/1701.00854, 2017.
- [3] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide, rev. 63. Technical report, October 2019.
- [4] Advanced Micro Devices. AMD64 Architecture Programmer’s Manual Volume 2: System Programming, rev 3.32. Technical report, October 2019.
- [5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. *Advances in computers*, 58(11):117–148, 2003.
- [6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.
- [7] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2, BtorMC and Boolector 3.0. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018.
- [8] Carsten Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Proc. of the 11th Intl. Conf. on Principles and Practice of Constraint Programming (CP 2005)*, pages 827–831, Sitges, Spain, October 2005.
- [9] Aina Niemetz and Mathias Preiner. Bitwuzla at the SMT-COMP 2020. *CoRR*, abs/2006.01621, 2020.

Appendices

A Intel Litmus Tests

Intel’s memory ordering litmus tests as seen in [3, Section 8.2.3] ported to our virtual machine model.

Neither Loads Nor Stores Are Reordered with Like Operations

The Intel-64 memory ordering model allows neither loads nor stores to be reordered with the same kind of operation. That is, it ensures that loads are seen in program order and that stores are seen in program order. This is illustrated by the following example:

Thread 0	Thread 1
ADDI 1 STORE 0 STORE 1	MEM 1 LOAD 0

Initial	Not Allowed ✗
<code>heap[0] = heap[1] = 0</code>	<code>mem₁ = 1 ∧ accu₁ = 0</code>

Example A.1: Stores Are Not Reordered with Other Stores [3, Example 8-1]

The disallowed return values could be exhibited only if thread 0’s two stores are reordered (with the two loads occurring between them) or if thread 1’s two loads are reordered (with the two stores occurring between them).

If `mem1 = 1`, the store to `heap[1]` occurs before the load from `heap[1]`. Because the Intel-64 memory ordering model does not allow stores to be reordered, the earlier store to `heap[0]` occurs before the load from `heap[1]`. Because the Intel-64 memory ordering model does not allow loads to be reordered, the store to `heap[0]` also occurs before the later load from `heap[0]`. Thus `accu1 = 1`.

Stores Are Not Reordered with Earlier Loads

The Intel-64 memory ordering model ensures that a store by a thread may not occur before a previous load by the same thread. This is illustrated by the following example:

Thread 0	Thread 1
MEM 0 ADDI 1 STORE 1	MEM 1 ADDI 1 STORE 0

Initial	Not Allowed ✗
heap[0] = heap[1] = 0	mem ₀ = mem ₁ = 1

Example A.2: Stores Are Not Reordered with Older Loads [3, Example 8-2]

Assume mem₀ = 1.

- Because mem₀ = 1, thread 1's store to heap[0] occurs before thread 0's load from heap[0].
- Because the Intel-64 memory ordering model prevents each store from being reordered with the earlier load by the same thread, thread 1's load from heap[1] occurs before its store to heap[0].
- Similarly, thread 0's load from heap[0] occurs before its store to heap[1].
- Thus, thread 1's load from heap[1] occurs before thread 0's store to heap[1], implying mem₁ = 0.

Loads May Be Reordered with Earlier Stores to Different Locations

The Intel-64 memory ordering model allows a load to be reordered with an earlier store to a different location. However, loads are not reordered with stores to the same location.

The fact that a load may be reordered with an earlier store to a different location is illustrated by the following example:

Thread 0	Thread 1
ADDI 1	ADDI 1
STORE 0	STORE 1
LOAD 1	LOAD 0

Initial	Allowed ✓
heap[0] = heap[1] = 0	accu ₀ = accu ₁ = 0

Example A.3: Loads May be Reordered with Older Stores [3, Example 8-3]

At each thread, the load and the store are to different locations and hence may be reordered. Any interleaving of the operations is thus allowed. One such interleaving has the two loads occurring before the two stores. This would result in each load returning value 0.

Loads Are Not Reordered with Older Stores to the Same Location

The Intel-64 memory ordering model allows a load to be reordered with an earlier store to a different location. However, loads are not reordered with stores to the same location.

The fact that a load may not be reordered with an earlier store to the same location is illustrated by the following example:

Thread 0	
ADDI 1 STORE 0 LOAD 0	
Initial	Not Allowed ✗
$\text{heap}[0] = 0$	$\text{accu}_0 = 0$

Example A.4: Loads Are not Reordered with Older Stores to the Same Location [3, Example 8-4]

The Intel-64 memory ordering model does not allow the load to be reordered with the earlier store because the accesses are to the same location. Therefore, $\text{accu}_0 = 1$ must hold.

Intra-Processor Forwarding Is Allowed

The memory ordering model allows concurrent stores by two threads to be seen in different orders by those two threads; specifically, each thread may perceive its own store occurring before that of the other. This is illustrated by the following example:

Thread 0	Thread 1
ADDI 1 STORE 0 LOAD 0 LOAD 1	ADDI 1 STORE 1 LOAD 1 LOAD 0

Initial	Allowed ✓
heap[0] = heap[1] = 0	accu ₀ = accu ₁ = 0

Example A.5: Intra-Processor Forwarding is Allowed [3, Example 8-5]

The memory ordering model imposes no constraints on the order in which the two stores appear to execute by the two threads. This fact allows thread 0 to see its store before seeing thread 1's, while thread 1 sees its store before seeing thread 0's. (Each thread is self consistent.) This allows $\text{accu}_0 = \text{accu}_1 = 0$.

In practice, the reordering in this example can arise as a result of store-buffer forwarding. While a store is temporarily held in a thread's store buffer, it can satisfy the thread's own loads but is not visible to (and cannot satisfy) loads by other threads.

Stores Are Transitively Visible

The memory ordering model ensures transitive visibility of stores; stores that are causally related appear to all threads to occur in an order consistent with the causality relation. This is illustrated by the following example:

Thread 0	Thread 1	Thread 2
ADDI 1 STORE 0	MEM 0 JNZ 3 ADDI 1 STORE 1	MEM 1 LOAD 0

Initial	Not Allowed ✗
<code>heap[0] = heap[1] = 0</code>	<code>mem₁ = mem₂ = 1 ∧ accu₂ = 0</code>

Example A.6: Stores Are Transitively Visible [3, Example 8-6]

Assume that `mem1 = 1` and `mem2 = 1`.

- Because `mem1 = 1`, thread 0's store occurs before thread 1's load.
- Because the memory ordering model prevents a store from being re-ordered with an earlier load (see [3, Section 8.2.3.3]), thread 1's load occurs before its store. Thus, thread 0's store causally precedes thread 1's store.
- Because thread 0's store causally precedes thread 1's store, the memory ordering model ensures that thread 0's store appears to occur before thread 1's store from the point of view of all threads.
- Because `mem2 = 1`, thread 1's store occurs before thread 2's load.
- Because the Intel-64 memory ordering model prevents loads from being reordered (see [3, Section 8.2.3.2]), thread 2's load occur in order.
- The above items imply that thread 0's store to `heap[0]` occurs before thread 2's load from `heap[0]`. This implies that `accu2 = 1`.

Stores Are Seen in a Consistent Order by Other Threads

As noted in [3, Section 8.2.3.5], the memory ordering model allows stores by two threads to be seen in different orders by those two threads. However, any two stores must appear to execute in the same order to all threads other than those performing the stores. This is illustrated by the following example:

Thread 0	Thread 1	Thread 2	Thread 3
ADDI 1 STORE 0	ADDI 1 STORE 1	MEM 0 LOAD 1	MEM 1 LOAD 0

Initial	Not Allowed ✗
heap[0] = heap[1] = 0	mem ₂ = mem ₃ = 1 ∧ accu ₂ = accu ₃ = 0

Example A.7: Stores Are Seen in a Consistent Order by Other Threads [3, Example 8-7]

By the principles discussed in [3, Section 8.2.3.2],

- thread 2's first and second load cannot be reordered,
- thread 3's first and second load cannot be reordered.
- If mem₂ = 1 and accu₂ = 0, thread 0's store appears to precede thread 1's store with respect to thread 2.
- Similarly, mem₃ = 1 and accu₃ = 0 imply that thread 1's store appears to precede thread 0's store with respect to thread 1.

Because the memory ordering model ensures that any two stores appear to execute in the same order to all threads (other than those performing the stores), this set of return values is not allowed.

Locked Instructions Have a Total Order

The memory ordering model ensures that all threads agree on a single execution order of all locked instructions. This is illustrated by the following example:

Thread 0	Thread 1	Thread 2	Thread 3
ADDI 1 CAS 0	ADDI 1 CAS 1	MEM 0 LOAD 1	MEM 1 LOAD 0

Initial	Not Allowed ✗
heap[0] = heap[1] = 0	mem ₂ = mem ₃ = 1 ∧ accu ₂ = accu ₃ = 0

Example A.8: Locked Instructions Have a Total Order [3, Example 8-8]

Thread 2 and thread 3 must agree on the order of the two executions of CAS. Without loss of generality, suppose that thread 0's CAS occurs first.

- If mem₃ = 1, thread 1's CAS into heap[1] occurs before thread 3's load from heap[1].
- Because the Intel-64 memory ordering model prevents loads from being reordered (see [3, Section 8.2.3.2]), thread 3's loads occur in order and, therefore, thread 1's CAS occurs before thread 3's load from heap[0].
- Since thread 0's CAS into heap[0] occurs before thread 1's CAS (by assumption), it occurs before thread 3's load from heap[0]. Thus, accu₃ = 1.

A similar argument (referring instead to thread 2's loads) applies if thread 1's CAS occurs before thread 0's CAS.

Loads Are Not Reordered with Locked Instructions

The memory ordering model prevents loads and stores from being reordered with locked instructions that execute earlier or later. The examples in this section illustrate only cases in which a locked instruction is executed before a load or a store. The reader should note that reordering is prevented also if the locked instruction is executed after a load or a store.

The first example illustrates that loads may not be reordered with earlier locked instructions:

Thread 0	Thread 1
ADDI 1	ADDI 1
CAS 0	CAS 1
LOAD 1	LOAD 0

Initial	Not Allowed ✗
heap[0] = heap[1] = 0	accu ₀ = accu ₁ = 0

Example A.9: Loads Are not Reordered with Locks [3, Example 8-9]

As explained in [3, Section 8.2.3.8], there is a total order of the executions of locked instructions. Without loss of generality, suppose that thread 0's CAS occurs first.

Because the Intel-64 memory ordering model prevents thread 1's load from being reordered with its earlier CAS, thread 0's CAS occurs before thread 1's load. This implies $\text{accu}_1 = 1$.

A similar argument (referring instead to thread 2's accesses) applies if thread 1's CAS occurs before thread 0's CAS.

Stores Are Not Reordered with Locked Instructions

The memory ordering model prevents loads and stores from being reordered with locked instructions that execute earlier or later. The examples in this section illustrate only cases in which a locked instruction is executed before a load or a store. The reader should note that reordering is prevented also if the locked instruction is executed after a load or a store.

The second example illustrates that a store may not be reordered with an earlier locked instruction:

Thread 0	Thread 1
ADDI 1 CAS 0 STORE 1	MEM 1 LOAD 0

Initial	Not Allowed X
$\text{heap}[0] = \text{heap}[1] = 0$	$\text{accu}_1 = 0 \wedge \text{mem}_1 = 1$

Example A.10: Stores Are not Reordered with Locks [3, Example 8-10]

Assume $\text{mem}_1 = 1$.

- Because $\text{mem}_1 = 1$, thread 0's store to $\text{heap}[1]$ occurs before thread 1's load from $\text{heap}[1]$.
- Because the memory ordering model prevents a store from being reordered with an earlier locked instruction, thread 0's **CAS** into $\text{heap}[0]$ occurs before its store to $\text{heap}[1]$.
- Thus, thread 0's **CAS** into $\text{heap}[0]$ occurs before thread 1's load from $\text{heap}[1]$.
- Because the memory ordering model prevents loads from being reordered (see [3, Section 8.2.3.2]), thread 1's loads occur in order and, therefore, thread 1's **CAS** into $\text{heap}[0]$ occurs before thread 1's load from $\text{heap}[0]$. Thus, $\text{accu}_1 = 1$.

B AMD Litmus Tests

AMD's memory ordering litmus tests as seen in [4, Section 7.2] ported to our virtual machine model.

Neither Loads Nor Stores Are Reordered with Like Operations

Successive stores from a single thread are committed to system memory and visible to other threads in program order. A store by a thread cannot be committed to memory before a read appearing earlier in the program has captured its targeted data from memory. In other words, stores from a thread cannot be reordered to occur prior to a load preceding it in program order.

Thread 0	Thread 1
ADDI 1 STORE 0 STORE 1	MEM 1 LOAD 0

Initial	Not Allowed X
$\text{heap}[0] = \text{heap}[1] = 0$	$\text{mem}_1 = 1 \wedge \text{accu}_1 = 0$

Example B.1: Stores Are Not Reordered with Other Stores [4, Example 1]

LOAD 0 cannot read 0 when **LOAD** 1 reads 1.

Stores Are Not Reordered with Earlier Loads

Successive stores from a single thread are committed to system memory and visible to other threads in program order. A store by a thread cannot be committed to memory before a read appearing earlier in the program has captured its targeted data from memory. In other words, stores from a thread cannot be reordered to occur prior to a load preceding it in program order.

Thread 0	Thread 1
MEM 0 ADDI 1 STORE 1	MEM 1 ADDI 1 STORE 0
Initial	Not Allowed ✗
heap[0] = heap[1] = 0	mem ₀ = mem ₁ = 1

Example B.2: Stores Are Not Reordered with Older Loads [4, Example 2]

LOAD 0 and LOAD 1 cannot both read 1.

Stores Can Be Arbitrarily Delayed

Stores from a thread appear to be committed to the memory system in program order; however, stores can be delayed arbitrarily by store buffering while the thread continues operation. Therefore, stores from a thread may not appear to be sequentially consistent.

Thread 0	Thread 1
ADDI 1 STORE 0 ADDI 1 STORE 0 LOAD 0	ADDI 1 STORE 1 ADDI 1 STORE 1 LOAD 1
Allowed ✓	
accu ₀ = accu ₁ = 1	

Example B.3: Stores Can Be Arbitrarily Delayed [4, Example 3]

Both LOAD 0 and LOAD 1 may read 1.

Loads May Be Reordered with Earlier Stores to Different Locations

Non-overlapping Loads may pass stores.

Thread 0	Thread 1
ADDI 1 STORE 0 LOAD 1	ADDI 1 STORE 1 LOAD 0

Initial	Allowed ✓
heap[0] = heap[1] = 0	accu ₀ = accu ₁ = 0

Example B.4: Loads May be Reordered with Older Stores [4, Example 4]

All combinations of values (00, 01, 10, and 11) may be observed by threads 0 and 1.

Sequential Consistency

Where sequential consistency is needed (for example in Dekker's algorithm for mutual exclusion), a **FENCE** instruction should be used between the store and the subsequent load, or an atomic instruction, such as **CAS**, should be used for the store.

Thread 0	Thread 1
ADDI 1 STORE 0 FENCE LOAD 1	ADDI 1 STORE 1 FENCE LOAD 0

Initial	Not Allowed ✗
heap[0] = heap[1] = 0	accu ₀ = accu ₁ = 0

Example B.5: Sequential Consistency [4, Example 5]

LOAD 0 and LOAD 1 cannot both read 0.

Stores Are Seen in a Consistent Order by Other Threads

Stores to different locations in memory observed from two (or more) other threads will appear in the same order to all observers. Behavior such as that is shown in this code example:

Thread 0	Thread 1	Thread 2	Thread 3
ADDI 1 STORE 0	ADDI 1 STORE 1	MEM 0 LOAD 1	MEM 1 LOAD 0

Initial	Not Allowed ✗
heap[0] = heap[1] = 0	mem ₂ = mem ₃ = 1 ∧ accu ₂ = accu ₃ = 0

Example B.6: Stores Are Seen in a Consistent Order by Other Threads [4, Example 6]

Thread 2 seeing STORE 0 from thread 0 before STORE 1 from thread 1, while thread 3 sees STORE 1 from thread 1 before STORE 0 from thread 0, is not allowed.

Stores Are Transitively Visible

Dependent stores between different threads appear to occur in program order, as shown in the code example below.

Thread 0	Thread 1	Thread 2
ADDI 1 STORE 0	MEM 0 JNZ 3 ADDI 1 STORE 1	MEM 1 LOAD 0

Initial	Not Allowed ✗
heap[0] = heap[1] = 0	mem ₁ = mem ₂ = 1 ∧ accu ₂ = 0

Example B.7: Stores Are Transitively Visible [4, Example 7]

If thread 1 reads a value from `heap[0]` (written by thread 0) before carrying out a store to `heap[1]`, and if thread 2 reads the updated value from `heap[1]`, a subsequent read of `heap[0]` must also be the updated value.

Intra-Processor Forwarding Is Allowed

The local visibility (within a thread) for a memory operation may differ from the global visibility (from another thread). Using a data bypass, a local load can read the result of a local store in a store buffer, before the store becomes globally visible. Program order is still maintained when using such bypasses.

Thread 0	Thread 1
ADDI 1	ADDI 1
STORE 0	STORE 1
MEM 0	MEM 1
LOAD 1	LOAD 0

Initial	Allowed ✓
heap[0] = heap[1] = 0	mem ₀ = mem ₁ = 1 ∧ accu ₀ = accu ₁ = 0

Example B.8: Intra-Processor Forwarding is Allowed [4, Example 8]

LOAD 0 in thread 0 can read 1 using the data bypass, while LOAD 0 in thread 1 can read 0. Similarly, LOAD 1 in thread 1 can read 1 while LOAD 1 in thread 0 can read 0. Therefore, the result mem₀ = 1, accu₀ = 0, mem₁ = 1 and accu₁ = 0 may occur. There are no constraints on the relative order of when the STORE 0 of thread 0 is visible to thread 1 relative to when the STORE 1 of thread 1 is visible to thread 0.

Global Visibility

If a very strong memory ordering model is required that does not allow local store-load bypasses, a **FENCE** instruction or an atomic instruction such as **CAS** should be used between the store and the subsequent load. This enforces a memory ordering stronger than total store ordering.

Thread 0	Thread 1
ADDI 1	ADDI 1
STORE 0	STORE 1
FENCE	FENCE
MEM 0	MEM 1
LOAD 1	LOAD 0

Initial	Not Allowed ✗
heap[0] = heap[1] = 0	mem ₀ = mem ₁ = 1 ∧ accu ₀ = accu ₁ = 0

Example B.9: Global Visibility [4, Example 9]

In this example, the **FENCE** instruction ensures that any buffered stores are globally visible before the loads are allowed to execute, so the result mem₀ = 1, accu₀ = 0, mem₁ = 1 and accu₁ = 0 will not occur.