Program Analysis for Weak Memory Models

Attila Zoltán Printz

printza@student.ethz.ch

Supervisor: Andrei Dan
Professor: Martin Vechev

Software Reliability Lab
Department of Computer Science
ETH Zurich

August 2015

Abstract

We present an approach to convert a program running under weak memory models (henceforth referred to as WMM) with possible non-deterministic qualities depending on the current memory ordering into a boolean program verifiable by our model checker. We will consider partial store order (PSO), and total store order (TSO) as possible WMMs. Abstracting our program to one operating on boolean variables representing our predicates, and then inputting this boolean program into a model checker, along with the correctness requirements specified for each program, is a feasible method to prove correctness, as shown by Ball et al. [2]. If we consider our program to apply a weak memory model however, this abstraction alone doesn't cover the additional complex behaviour exhibited by WMM. Because of this, we employ a method called buffer size analysis, combined with the technique described by Dan et al. [1], in order to achieve a sound boolean program abstraction of any program employing one of the memory ordering methods we consider.

1 Introduction

A vital use of program analysis is correctness proving. Assuming we have a program and a set of criteria our program has to meet under any circumstance, proving that our program meets all of our criteria is an important task for critical software components. Additionally, if we expand our validity requirements to concurrent runs of one or more instances of the program, which itself might have several concurrent processes, our analysis threatens to become untractably complex. When we relax the constraints of sequential consistency to

1

those defined by total store ordering (TSO) and partial store ordering (PSO) for programs employing weak memory models (WMM), we face a seemingly untractable problem when trying to perform program analysis.

1.1 Proof of correctness

Let P be a multi-process program using a WMM instruction set, let E be a list of predicates on the variables of P, and let a be a boolean expression describing the correctness assertions for P using the predicates provided in E and the program counter. We consider P to be correct as defined by E, iff for every possible run scenario of any number of concurrent instances of P, in which any number of processes of P may run concurrently, the assertion a holds true in every program point of P. Terms involving the program counter are used to limit the necessity for a (or parts thereof) to hold true. By abstracting P as a boolean program using E, such a proof becomes feasible using a model checker.

1.2 Weak memory models

In all of our programs, we will differentiate between local resources, declared as local variables, and their remote counterparts. The main feature of WMM languages is the differentiation between local assignments and remote reads and writes. We consider local variables as being local both in the semantic sense (i.e. a process constitutes the scope of a local variable) and the technical sense (i.e. a local variable is stored in fast-access memory, such as a register or local cache space). In contrast, remote variables are globally shared across all processes and are assumed to be stored in shared memory, that's more time-consuming to access. We will continue using the words "local" and "remote" in this manner throughout this work.

Thus, the main advantage of WMM programming becomes clear, as the programmer will now be able to avoid costly remote memory accesses for operations restricted to one process, along with any adverse side-effects depending on the caching policy. This does not, however, mean that a remote statement will always access remote memory. For practical reasons, remote writes are immediately stored in a buffer, and only committed to remote memory in the case of a flush statement, which, as apparent from its definition, is inherently non-deterministic. In the same vein, remote reads will read from the local buffer if possible, and only access remote memory if the local buffer is empty. In this work, we will abstract the buffer as a series of local variables representing a FIFO queue. The buffering method depends on the memory model. In the case of PSO, every global variable in every process will have a dedicated buffer, whereas in the case of TSO, every process will have one buffer shared by all global variables.

1.3 Difficulties of WMM

Our program behaves differently depending on the memory ordering chosen, without this difference being readily apparent from the code. More precisely, when using PSO, the available buffer size defines a set maximum buffer size for every global variable per process, whereas when using TSO, it only defines the sum of the buffer sizes of every global variable per process. To illustrate this with an example, if we have a program with 2 global variables x and y, with 4 remote writes in x and 2 remote writes in y before flushing, a per-process buffer size of 6 wouldn't result in a buffer overflow, as the sum of the maximum buffer sizes of x and y does not exceed 6. However, the same code results in an overflow if we define a maximum per-process per-variable buffer size of 3, since the 4th write in x would exceed the maximum buffer size allocated to x in that process.

We tackle this problem by converting our program P from one generally exploiting all the features available in WMM into another program P' that lacks any "true" WMM statements. I.e., our resulting program will only contain local assignments, remote reads, and remote writes, with no fences or flushes (we will redefine flushes as statements later on). Though remote reads and writes will still be featured in P', they will simply describe immediate and deterministic remote access statements, without any of the advantages or pitfalls of their WMM counterparts. Finally, we can convert P' into a boolean program using predicate abstraction, which, as described in 1.1, can be used to prove the correctness of P regarding a.

Flushing is a part of WMMs. It describes an operation triggered by memory management at times usually out of the programmer's reach. In essence, flushing writes the buffer content to memory, deleting it from the buffer. Since the behaviour of flushing is an important aspect of WMMs to consider when attempting to analyse programs running under them, we need to be able to simulate its behaviour in a non-WMM environment. To this end, we define `flush;` to be an explicit statement in our program, and we will place this statement once wherever a flush might occur.

1.4 Main contributions

This work describes the implementation of a software tool that takes a program written in SALPL as input, a language we define below, and outputs a boolean program parsable by the Fender [4] model checker for verification. The source code and software is available at `https://github.com/hetmeter/awmm` . The components of this tool break down as follows:

Parser We have built a lightweight parser that takes a syntactically correct SALPL program and a language definition of SALPL as input, and outputs

an abstract semantic tree representation into a file. This specific form of AST representation is then used to perform further operations on the program.

Buffer size analysis   We traverse the control flow tree of the program, and record the number of store and fence statements we encounter on our path. For every store statement, we increment the buffer size needed to accomodate all stores so far. For PSO programs, this is stored separately for each process and each variable in a value to which we will refer as $s_{x,i}$ for the variable x and the process i. For TSO programs, this value is only kept by process, independent of variables. Analogously to its PSO counterpart, we will refer to this value as s_i. For buffer sizes exceeding K, as well as for values that may be arbitrarily high (which we will indicate by assigning the value "TOP"), we will store the value K.

Program rewriting   Converting the WMM program to one lacking buffer behaviour influence is done by visiting every program point of the type store, load, fence, or flush. For every program point, we will have already computed all parameters needed to generate the replacement code. In this phase, we carry out the replacements we define in 3.3.

Predicate abstraction   We take our rewritten program, which is now devoid of any operations on the buffer. In this phase, our tool generates tables for every weakest liberal precondition and implicative cube disjunction we use for the predicate abstraction of our program. These components and techniques are expanded upon in section 4. We use Z3 [5] as a theorem prover to prove or disprove implications. As a general rule, we try to minimize the calls to the theorem prover, and we do this with a combination of storing all computed results for later use, omitting cubes that form supersets of already found implicative cubes, and renaming heuristics, with which we reduce the problem of finding cube disjunctions for predicates involving derived variables to the same problem involving the global variables from which they were derived.

Model checking   While not strictly a part of this work, our output is tailored to be used with the model checker Fender [4]. Our output can be directly fed into the input of Fender for analysis.

2   Programming language

We define a programming language that is both simple and powerful enough to conclusively describe weak memory models to the extent they may be available to the programmer. Throughout this work, we will refer to this language as SALPL, abbreviating the first description of the "simple assembly-like programming language" [3] defined by Kuperstein, Vechev and Yahav in [3], on which SALPL is based. A SALPL program has only integer-type variables, but

it can perform comparisons that return boolean values, which in turn can be operated on using boolean operators. Boolean values are used in conditionals and may have the values *true*, *false*, or $*$, all of which are also valid literals. Comments are defined as in C, with `//` prepending line comments and `/*` and `*/` enclosing block comments.

In this section, we will group code placeholders using [ ]-brackets, which might have a * appended for an arbitrary number of the code specified.

Accepting program state A valid SALPL program must have the following structure:

```
beginit
     [store statement]*
endinit

[process declaration]*

[assertion]
```

Assertion The assertion may be appended to the end of the program. Its syntax is derived from the boolean program syntax used by the Fender [4] model checker. The boolean expression therein may only refer to global variables, constants, or program counters of the form `pc(`i`)`, with i being a valid process index. The value of a program counter may be that of any valid label of the process i.

```
assert(always(boolean expression))
```

Process declaration Every process must have a unique integer identifier i.

```
process i:
     [statement]*
```

Statement A *statement* can be any of the following: *store statement*, *load statement*, *local assignment statement*, *label*, *goto statement*, *flush statement*, *fence statement*, *abort statement*, *nop statement*, *if-else block*.

Store statement Generally, the store statement writes the right-hand side expression to the buffer, pending commitment to remote memory. In the special case where flushes are to be excluded, as it will be the case later on in this work, the store statement may be viewed as an immediate and deterministic remote write operation. The left-hand side identifier must be a global variable.

```
store variable identifier = integer expression;
```

Load statement   Generally, the load statement reads the most recent relevant value from the buffer, and it only reads from remote memory if the buffer doesn't contain any values of the desired variable. In the special case where flushes are to be excluded, as it will be the case later on in this work, the load statement may be viewed as an unconditional remote read operation. The left-hand side identifier must be a local variable, while the right-hand side identifier must be a global variable.

```
load  variable identifier  =  variable identifier ;
```

Local assignment statement   The left-hand side identifier must be a local variable. The *integer expression* may be an integer literal, an integer variable symbol, or any integer operation on them.

```
variable identifier  =  integer expression ;
```

Label   The statement following the colon may not be another label. The label value i must be unique within its process.

```
i:  statement
```

Goto statement   The label value i must be a valid label within the calling process.

```
goto  i ;
```

Flush statement   Flushing will iterate through all global variables and non-deterministically commit the first value of the respective variable to remote memory and remove that value from the buffer. If a variable does not have any values to be committed, it will be ignored.

```
flush ;
```

Fence statement   Without any operations on the buffer or memory, fences just assume the buffer to be empty from then on. If the buffer is not empty, the fence blocks until it becomes empty.

```
fence ;
```

Abort statement   Aborts the program with the *message* argument as the error message.

```
abort("message");
```

Nop statement Does nothing.

```
nop;
```

If-else block If the *boolean expression* evaluates to *true*, the *statement* block right after the conditional is carried out, omitting the *else block*, if it exists. If it evaluates to *false*, jumps to the point after the first *statement* block. If it evaluates to ∗, i.e. *undecided*, it will non-deterministically simulate an evaluation to either *true*, or *false*.

```
if  (boolean expression)
      [statement]*
[else block]?
endif;
```

with the *else block* being:

```
else
      [statement]*
```

3   Buffer abstraction

As we have seen, some of the statements available to the programmer make use of a write buffer. Since depending on the memory model chosen, we may encounter different behaviours when running the same program using different memory models, we face the challenge of rewriting the program in a way that its behaviour will be conclusively defined, independent of the memory model chosen. We will do this by completely eliminating any buffer access by abstracting the buffers as sets of local variables. In order for us to be able to assume the lack of buffer operations, we will need to eliminate fence- and flush statements. As for remote stores and loads, we can replace them with code that simulates their buffer-related behaviours using non-WMM statements, retaining the use of store and load statements to represent assignment statements involving global variables without the involvement of the buffer. Since this intermediary code will never be run, but instead will be further abstracted in the predicate abstraction stage, we may safely ignore any regular WMM-behaviour such statements might have, as we will handle store- and load statements like local assignments for predicate abstraction purposes. The reason we refrain from using local assignment statements instead of store- or load statements in our abstracted program is our desire to preserve syntactic correctness even in our intermediary program, and to highlight the fact, that at certain points, load statements will actually perform a remote read, and that flush statements may at one point actually perform remote writes.

3.1 Buffer size analysis

For the purpose of determining the maximum buffer space needed by any global variable, we count the number of store statements in the control flow of the program within each process i. In case there are less store statements on a variable than the user-specified maximum buffer size K, we will only allocate the necessary number of buffer variables, which is the minimum of K and the number of stores on that variable in the process i. We will refer to this number as $s_{x,i}$ for any global variable x when employing PSO, and s_i for TSO.

In detail, we first take the AST representing the program, and cascade through it from the root to the leaves. Whenever we encounter a program point, we draw one control flow edge from that program point to any other program point that might directly follow in a possible sequential execution order. As shown in Figure 1, keywords such as `else` and `endif` are not program points, yet `nop` or unlabelled statements are.

Once we have the control flow graph, we start at the first program point in every process i with a visitor having s_i or $s_{x,i}$ for all global variables x set to 0, and traverse the entire graph until we encounter a dead end, or an already visited node (which is the case for cyclic control flow graphs). Every time we visit a program point, we take the maxima of the visitor's $s_{x,i}$ or s_i, depending on the current store order, and the current node's corresponding values, and store them in both the node and the visitor. If the visited node is a store, we increment the corresponding values by 1 beforehand. Figure 2. shows the snapshot of the state of the program with a visitor just having visited the third node from the top.

Note that all buffer size entries are initialized to 0. The visitor propagates the accumulated values to every subsequent node visited. If a visitor encounters a node with more than one outgoing edge, it spawns a copy of itself for every additional edge. These copies carry the same values as their predecessors, and they have the same record of visited nodes, i.e. if a visitor comes upon a node that had already been visited by any one of its ancestors, it will also regard that node as visited. Conversely, if a visitor encounters a node which had been visited by at least one other visitor, but none of its direct ancestors, it will not regard that node as visited, and it will continue down the control flow graph as usual.

If a visitor encounters a node it regards to be visited, it still increments the corresponding value as needed. At this point, if there is any value stored in the node that is smaller than its counterpart stored by the visitor, an arbitrary increase of that value is detected and that value is set to TOP (which is regarded as infinitely large for comparison reasons). The node communicates this change in the direction of the control flow, and every node that is situated downward of this current node also gets its corresponding value set to TOP. At this point,

8

the visitor stops. Figure 3. shows the snapshot of the state of the program with a visitor just having revisited the third node from the top, Figure 4. shows the aftermath of the TOP values cascading along the control flow edges, and finally, Figure 5. shows the state after the last visitor reaches the bottom most node after having been spawned in the fourth node from the top.

Note that while the order of evaluating the path of multiple visitors is irrelevant, their affecting each other by node value editing creates a race condition on nodes that are potentially visited by more than one visitor. Thus, the buffer size analysis is not parallelizable on the same process. However, different processes may be examined in parallel.
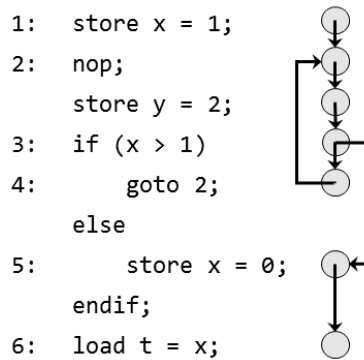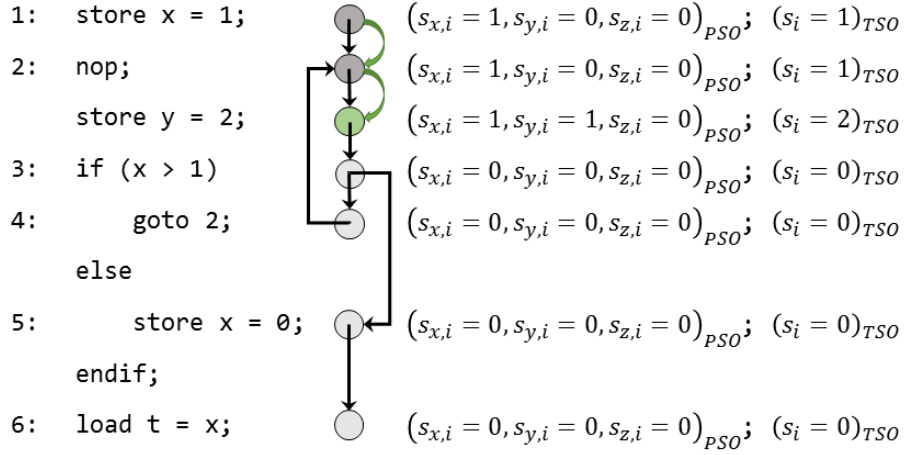
Figure 1: Control flow example

```
1:   store x = 1;

2:   nop;
     store y = 2;

3:   if (x > 1)
4:       goto 2;
     else
5:       store x = 0;
     endif;
6:   load t = x;
```

Figure 2: Visitor reached 3rd node

```
1:    store x = 1;
2:    nop;
      store y = 2;
3:    if (x > 1)
4:        goto 2;
      else
5:        store x = 0;
      endif;
6:    load t = x;
```

$(s_{x,i} = 1, s_{y,i} = 0, s_{z,i} = 0)_{PSO};\ (s_i = 1)_{TSO}$

$(s_{x,i} = 1, s_{y,i} = 0, s_{z,i} = 0)_{PSO};\ (s_i = 1)_{TSO}$

$(s_{x,i} = 1, s_{y,i} = 1, s_{z,i} = 0)_{PSO};\ (s_i = 2)_{TSO}$

$(s_{x,i} = 0, s_{y,i} = 0, s_{z,i} = 0)_{PSO};\ (s_i = 0)_{TSO}$

$(s_{x,i} = 0, s_{y,i} = 0, s_{z,i} = 0)_{PSO};\ (s_i = 0)_{TSO}$

$(s_{x,i} = 0, s_{y,i} = 0, s_{z,i} = 0)_{PSO};\ (s_i = 0)_{TSO}$

$(s_{x,i} = 0, s_{y,i} = 0, s_{z,i} = 0)_{PSO};\ (s_i = 0)_{TSO}$

Figure 3: Visitor reached 3rd node again

```
1:    store x = 1;
2:    nop;
      store y = 2;
3:    if (x > 1)
4:        goto 2;
      else
5:        store x = 0;
      endif;
6:    load t = x;
```

$(s_{x,i} = 1, s_{y,i} = 0, s_{z,i} = 0)_{PSO};\ (s_i = 1)_{TSO}$

$(s_{x,i} = 1, s_{y,i} = \mathsf{T}, s_{z,i} = 0)_{PSO};\ (s_i = \mathsf{T})_{TSO}$

$(s_{x,i} = 1, s_{y,i} = 1, s_{z,i} = 0)_{PSO};\ (s_i = 2)_{TSO}$

$(s_{x,i} = 1, s_{y,i} = 1, s_{z,i} = 0)_{PSO};\ (s_i = 2)_{TSO}$

$(s_{x,i} = 1, s_{y,i} = 1, s_{z,i} = 0)_{PSO};\ (s_i = 2)_{TSO}$

$(s_{x,i} = 0, s_{y,i} = 0, s_{z,i} = 0)_{PSO};\ (s_i = 0)_{TSO}$

$(s_{x,i} = 0, s_{y,i} = 0, s_{z,i} = 0)_{PSO};\ (s_i = 0)_{TSO}$

10
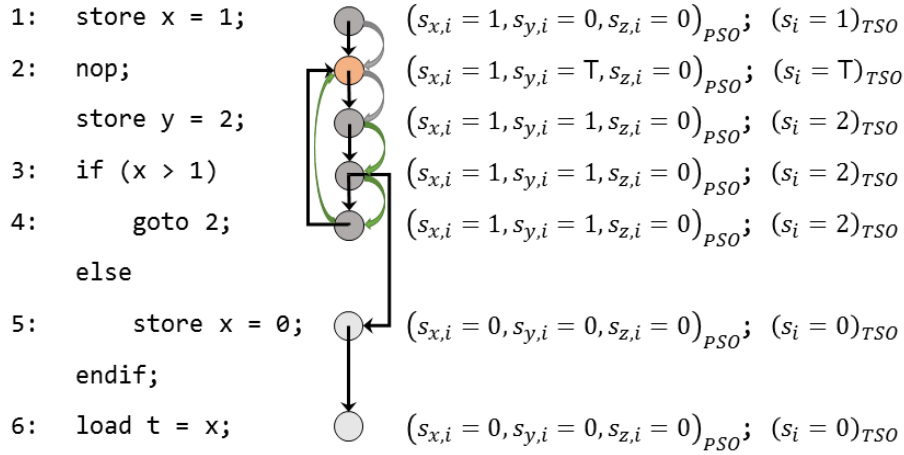
Figure 4: TOP values cascaded

1:	store x = 1;	$\left(s_{x,i} = 1, s_{y,i} = 0, s_{z,i} = 0\right)_{PSO}$; $(s_i = 1)_{TSO}$
2:	nop;	$\left(s_{x,i} = 1, s_{y,i} = \mathsf{T}, s_{z,i} = 0\right)_{PSO}$; $(s_i = \mathsf{T})_{TSO}$
	store y = 2;	$\left(s_{x,i} = 1, s_{y,i} = \mathsf{T}, s_{z,i} = 0\right)_{PSO}$; $(s_i = \mathsf{T})_{TSO}$
3:	if (x > 1)	$\left(s_{x,i} = 1, s_{y,i} = \mathsf{T}, s_{z,i} = 0\right)_{PSO}$; $(s_i = \mathsf{T})_{TSO}$
4:	goto 2;	$\left(s_{x,i} = 1, s_{y,i} = \mathsf{T}, s_{z,i} = 0\right)_{PSO}$; $(s_i = \mathsf{T})_{TSO}$
	else	$\left(s_{x,i} = 0, s_{y,i} = \mathsf{T}, s_{z,i} = 0\right)_{PSO}$; $(s_i = \mathsf{T})_{TSO}$
5:	store x = 0;	$\left(s_{x,i} = 0, s_{y,i} = \mathsf{T}, s_{z,i} = 0\right)_{PSO}$; $(s_i = \mathsf{T})_{TSO}$
	endif;	
6:	load t = x;	$\left(s_{x,i} = 0, s_{y,i} = \mathsf{T}, s_{z,i} = 0\right)_{PSO}$; $(s_i = \mathsf{T})_{TSO}$

Figure 5: Visitor reached last node

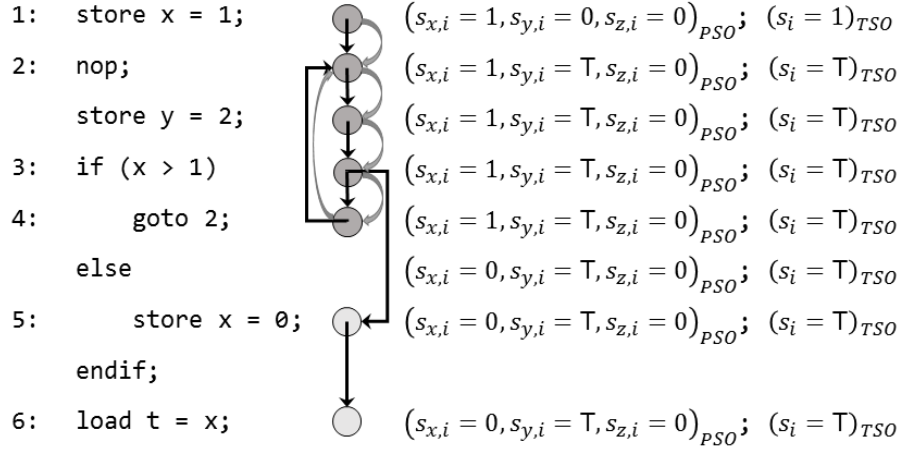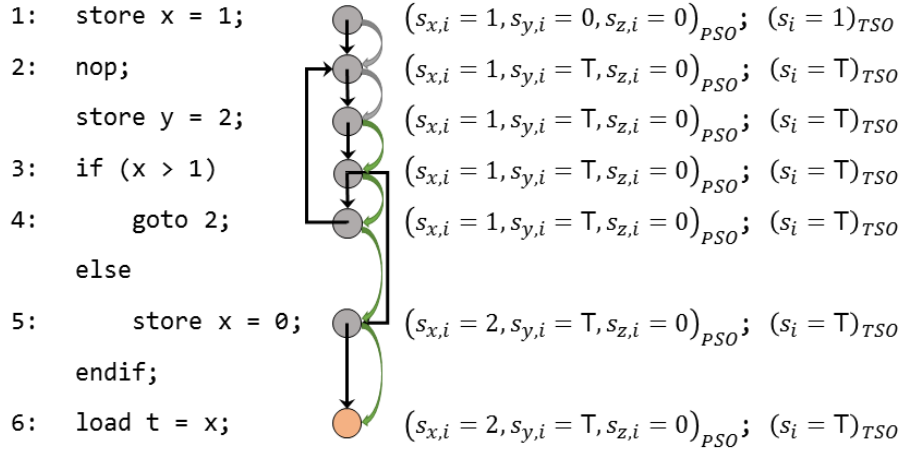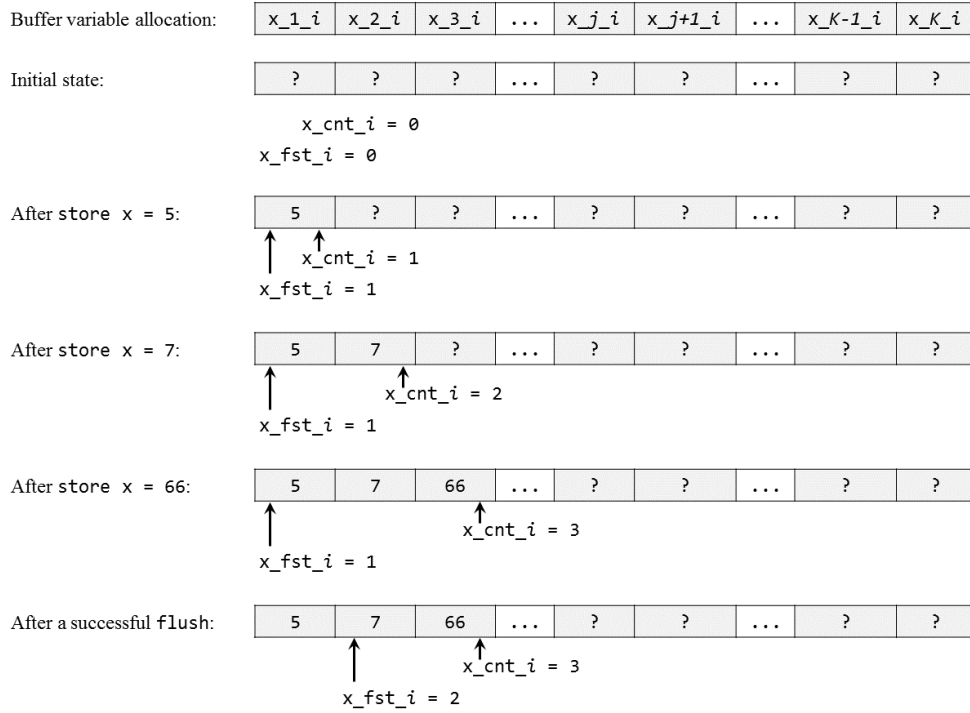|		
1:	store x = 1;	$\left(s_{x,i} = 1, s_{y,i} = 0, s_{z,i} = 0\right)_{PSO}$; $(s_i = 1)_{TSO}$
2:	nop;	$\left(s_{x,i} = 1, s_{y,i} = \mathsf{T}, s_{z,i} = 0\right)_{PSO}$; $(s_i = \mathsf{T})_{TSO}$
	store y = 2;	$\left(s_{x,i} = 1, s_{y,i} = \mathsf{T}, s_{z,i} = 0\right)_{PSO}$; $(s_i = \mathsf{T})_{TSO}$
3:	if (x > 1)	$\left(s_{x,i} = 1, s_{y,i} = \mathsf{T}, s_{z,i} = 0\right)_{PSO}$; $(s_i = \mathsf{T})_{TSO}$
4:	goto 2;	$\left(s_{x,i} = 1, s_{y,i} = \mathsf{T}, s_{z,i} = 0\right)_{PSO}$; $(s_i = \mathsf{T})_{TSO}$
	else	
5:	store x = 0;	$\left(s_{x,i} = 2, s_{y,i} = \mathsf{T}, s_{z,i} = 0\right)_{PSO}$; $(s_i = \mathsf{T})_{TSO}$
	endif;	
6:	load t = x;	$\left(s_{x,i} = 2, s_{y,i} = \mathsf{T}, s_{z,i} = 0\right)_{PSO}$; $(s_i = \mathsf{T})_{TSO}$

11

3.2 PSO abstraction

Using PSO, every global variable has a separate buffer per process of the size $s_{x,i}$. Therefore, for each process i, for each global variable x, there are $s_{x,i}$ buffer variables of the form x_j_i, where $j \in [1, s_{x,i}]$. Since flushes are non-deterministic, we cannot keep track of occupied buffer spaces. Therefore, we declare 2 additional auxiliary variables for each global variable in each process: x_cnt_i, which is initialised to 0 and otherwise has the value of the last buffer index to which a value of x was written, and x_fst_i, which is analogous to x_cnt_i with the difference that it points to the first such index. Figure 6. illustrates the buffer allocation abstraction we apply for PSO programs.

Figure 6: PSO buffer abstraction approach

	x_1_i	x_2_i	x_3_i	...	x_j_i	x_j+1_i	...	x_K-1_i	x_K_i
Buffer variable allocation:	x_1_i	x_2_i	x_3_i	...	x_j_i	x_j+1_i	...	x_K-1_i	x_K_i

Initial state:

?	?	?	...	?	?	...	?	?

x_cnt_i = 0
x_fst_i = 0

After store x = 5:

5	?	?	...	?	?	...	?	?

x_cnt_i = 1
x_fst_i = 1

After store x = 7:

5	7	?	...	?	?	...	?	?

x_cnt_i = 2
x_fst_i = 1

After store x = 66:

5	7	66	...	?	?	...	?	?

x_cnt_i = 3
x_fst_i = 1

After a successful flush:

5	7	66	...	?	?	...	?	?

x_cnt_i = 3
x_fst_i = 2

3.3 TSO abstraction

Using TSO, every process has a separate buffer of the size K, which all global variables share. The buffer holds all values in an order-preserving manner: A value v_1 that is stored in the buffer before another value v_2 will also be written to memory before v_2. Therefore, the store order of values must be preserved in the program.

An economic way to do this is to keep track of the last element inserted into the buffer, and store the next value in the following slot. Since we cannot know the actual run-time allocation of the buffer, we will only be able to analyse the maximum buffer size of the entire process (which we express by s_i), but not of the individual variables. For each buffer slot, we will declare two variables of the form `buf_j_i`, and `own_j_i`, where $j \in [1, s_i]$.

We will also statically build an allocator table, which will pair every global variable to a unique numeric value which will allow us to identify the values stored in the buffer as belonging to its owner variable at run-time. We will use the notation $\rho(v)$ to denote the numeric allocator key of the variable `v`. Thus, for every buffer slot with the number $j \in [1, s_i]$ in every process i, we will have a variable `buf_j_i` (which we will call a buffer variable) containing the value currently held by that buffer slot, and a variable `own_j_i` (which we will call a buffer allocator) containing the allocator key of the variable whose value is stored there. If the allocator is set to 0, the buffer is marked to be free, and if it is set to -1, it is marked as invalidated by a flush. Figure 7. illustrates the buffer allocation abstraction we apply for TSO programs.

Figure 7: TSO buffer abstraction approach

Buffer variables:	buf_1_i	buf_2_i	...	buf_j_i	buf_j+1_i	...	buf_K-1_i	buf_K_i
Buffer allocators:	own_1_i	own_2_i	...	own_j_i	own_j+1_i	...	own_K-1_i	own_K_i

```
1: store x = 8;
2: store y = 3;
3: if (*) goto 2;
4: store x = -1;
5: store y = 7;
6: flush;
```

Legend: nonvariable representations

Allocator value (interpretation)	Allocator key (numerical code)
not allocated	0
flushed	-1

Buffer allocator values

Allocator value (variable name)	Allocator key (numerical representation)
x	1
y	2

Initial state

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...	K
Buffer variables:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0
Buffer allocators:	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?		

After store x = 8;

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...	K
Buffer variables:	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0
Buffer allocators:	1	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?		

After store y = 3; for the 1st time

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...	K
Buffer variables:	8	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0
Buffer allocators:	1	2	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?		

After store y = 3; for the nth time

	1	2	3	...	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	...	K
Buffer variables:	8	3	3	...	3	3	0	0	0	0	0	0	0	...	0
Buffer allocators:	1	2	2		2	2	?	?	?	?	?	?	?		

After store x = -1; after n-1 jumps to 2:

	1	2	3	...	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	...	K
Buffer variables:	8	3	3	...	3	3	-1	0	0	0	0	0	0	...	0
Buffer allocators:	1	2	2		2	2	1	?	?	?	?	?	?		

After store y = 7;

	1	2	3	...	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	...	K
Buffer variables:	8	3	3	...	3	3	-1	7	0	0	0	0	0	...	0
Buffer allocators:	1	2	2		2	2	1	2	?	?	?	?	?		

After flush; having flushed the first 2 buffer slots

	1	2	3	...	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	...	K
Buffer variables:	8	3	3	...	3	3	-1	7	0	0	0	0	0	...	0
Buffer allocators:	-1	-1	2		2	2	1	2	?	?	?	?	?		

14

3.4 Replacement rules

After we compute all $s_{x,i}$ or s_i for each global variable x and for each process i, we have enough data to statically perform all necessary replacements in our program to allow predicate abstraction. We will define different replacement rules for each store order.

3.4.1 Replacing store statements

Assuming x is a global variable, r is an integer expression, and we encounter store x = r in a process i, we will replace this statement with the code blocks listed below. Note, that the variable symbol is appended to every corresponding if-else statement in the case of TSO. This is done in order to allow us a fast context analysis on buffer variable assignments within these if-else blocks, which we will use later on.

PSO approach:

```
if (x_cnt_i = s_{x,i})
    abort("overflow");
endif;

if (x_fst_i = 0)
    x_fst_i = 1;
    x_1_i = r;
endif;

x_cnt_i = x_cnt_i + 1;

if (x_cnt_i = 2)
    x_2_i = r;
endif;
...
if (x_cnt_i = j)
    x_j_i = r;
endif;
...
if (x_cnt_i = s_{x,i})
    x_s_{x,i}_i = r;
endif;
```

TSO approach:

```
if (own_1_i = 0)     // x
    buf_1_i = r;
    own_1_i = ρ(x);
    goto SEL;
endif;

if (own_2_i = 0)     // x
    buf_2_i = r;
    own_2_i = ρ(x);
    goto SEL;
endif;
...
if (own_j_i = 0)     // x
    buf_j_i = r;
    own_j_i = ρ(x);
    goto SEL;
endif;
...
if (own_s_i_i = 0)    // x
    buf_s_i_i = r;
    own_s_i_i = ρ(x);
    goto SEL;
endif;
abort("overflow");
SEL: nop;
```

15

3.4.2 Replacing load statements

Assuming x is a global variable, l is a local variable, and we encounter load l = x in a process i, we will replace this statement with:

PSO approach:

```
if (x_cnt_i = 0)
    load l = x;
endif;

if (x_cnt_i = s_{x,i})
    l = x_s_{x,i}_i;
endif;
...
if (x_cnt_i = j)
    l = x_j_i;
endif;
...
if (x_cnt_i = 2)
    l = x_2_i;
endif;

if (x_cnt_i = 1)
    l = x_1_i;
endif;
```

TSO approach:

```
if (own_s_i_i = ρ(x)) // x
    l = buf_s_i_i;
    goto LEL;
endif;
...
if (own_j_i = ρ(x)) // x
    l = buf_j_i;
    goto LEL;
endif;
...
if (own_2_i = ρ(x)) // x
    l = buf_2_i;
    goto LEL;
endif;

if (own_1_i = ρ(x)) // x
    l = buf_1_i;
    goto LEL;
endif;
load l = V;
LEL: nop;
```

3.4.3 Replacing fence statements

Assuming we encounter fence; in a process i, we will replace this statement with the following block for every global variable x:

PSO approach:

```
assume (x_cnt_i = 0);
assume (x_fst_i = 0);
```

TSO approach:

```
assume (own_1_i < 1);
assume (own_2_i < 1);
...
assume (own_j_i < 1);
...
assume (own_s_i_i < 1);
own_1_i = 0;
own_2_i = 0;
...
own_j_i = 0;
...
own_s_i_i = 0;
```

3.4.4 Replacing flush statements

Let β be the number of global variables in P and ρ^{-1} be the inverse function of ρ, i.e. it returns the variable symbol belonging to the input number. Assuming we encounter `flush;` in a process i, we will replace this statement with:

TSO approach:

```
if (*)
    goto FEL;
endif;
if (own_1_i > 0)
    [flush 1]
endif;
if (*)
    goto FEL;
endif;
if (own_2_i > 0)
    [flush 2]
endif;
...
if (*)
    goto FEL;
endif;
if (own_j_i > 0)
    [flush j]
endif;
...
if (*)
    goto FEL;
endif;
if (own_s_i_i > 0)
    [flush s_i]
endif;
FEL: nop;
```

The placeholder $[flush\ \alpha]$ describes the following code block for a number α:

```
if (own_α_i = 1) // ρ⁻¹(1)
    store ρ⁻¹(1) = buf_α_i;
    goto SubFEL;
endif;
if (own_α_i = 2) // ρ⁻¹(2)
    store ρ⁻¹(2) = buf_α_i;
    goto SubFEL;
endif;
...
if (own_α_i = j) // ρ⁻¹(j)
    store ρ⁻¹(j) = buf_α_i;
    goto SubFEL;
endif;
...
if (own_α_i = β) // ρ⁻¹(β)
    store ρ⁻¹(β) = buf_α_i;
    goto SubFEL;
endif;
SubFEL: own_α_i = -1;
```

PSO approach:

```
p:  if (*)
        [global variable flushes]
        goto p;
endif;
```

The placeholder [*global variable flushes*] describes the following code block for each global variable x:

```
if (x_cnt_i > 0)
    if (*)
        if (x_fst_i > 1)
            if (x_fst_i > 2)
                ...
                if (x_fst_i > j)
                    ...
                    if (x_fst_i > s_{x,i} - 1)
                        store x = x_s_{x,i}_i;
                    else
                        store x = x_(s_{x,i} - 1)_i;
                    endif;
                    ...
                else
                    store x = x_j_i;
                endif;
                ...
            else
                store x = x_2_i;
            endif;
        else
            store x = x_1_i;
        endif;

        x_fst_i = x_fst_i + 1;
    endif;
endif;
```

4 Predicate abstraction

After the replacements carried out in the buffer size analysis section, we have a program that both encapsulates the behaviour of the chosen WMM, and that can be abstracted according to the rules described in section 4 up and until 4.4 of [2]. As our program is not written in C, but in SALPL, and the output language must be recognized by our model checker, Fender, we state the following alterations to the aforementioned rules: there are no procedure calls, pointers,

arrays, or any variables of a type other than integer in our input program, and no variables of a type other than boolean in the output. In this section, we define the components, and lastly, the overall structure of our output boolean program. We will use the notation $\sigma[\texttt{a}/\texttt{b}]$ to denote σ with all occurences of $\texttt{b}$ replaced by $\texttt{a}$.

For the predicate abstraction itself, we declare two boolean variables for each predicate $\varphi_i \in E$: $\texttt{B\_}i$ (boolean variables) and $\texttt{T\_}i$ (temporary variables). Both of them hold the truth value of φ_i, albeit at different times. We can guarantee however, that $\texttt{B\_}i$ is always the truth value of φ_i. $\texttt{T\_}i$ on the other hand is used during parallel assignments, where we simultaneously assign values to more than one boolean variable, while still needing to read values thereof from before the commencing of the parallel assignments. These variables are also declared for the additional predicates defined below. Also, keeping in line with the notation in the paper by Ball et al. [2], we will use the notation $WP(\texttt{s}, \varphi)$ for the weakest liberal precondition of the predicate φ with respect to the statement s, and $\mathcal{F}_V(\varphi)$ for the largest disjunction of implicative cubes for a predicate φ over a set of boolean variables V.

4.1 Variables and predicates

Our program has a set of variables H, as declared in the original input program, and a set of predicates E over H. The predicates in E are sufficient to verify the program under sequential consistency constraints (i.e. without WMM-related effects). Among the newly declared local variables, i.e. all variables that weren't part of the original program, but since have been declared during the buffer size analysis stage, we differentiate between auxiliary variables, which encompass all first pointers, counters, and buffer allocators of the forms $\texttt{x\_fst\_}i$, $\texttt{x\_cnt\_}i$, and $\texttt{own\_x\_}i$, respectively, and buffer variables, which describe all variables abstracting WMM write buffer data. All buffer variables are assigned to a single global variable at any program point. This follows from the buffer variables' inherent association with a single global variable in the case of PSO, and TSO buffer variables' only being used after allocation to a global variable. With this information, for each global variable $\texttt{x}$ and each valid process number i, we extend our predicate set by the following predicates:

In the case of PSO:

```
( x _ c n t _ i  >=  0 )
( x _ c n t _ i  <=  K )
( x _ f s t _ i  >=  0 )
( x _ f s t _ i  <=  K )
```

In the case of TSO:

```
( o w n _ 1 _ i  =  ρ(x) )
( o w n _ 1 _ i  =  ρ(x) )
...
( o w n _ j _ i  =  ρ(x) )
...
( o w n _ s_i _ i  =  ρ(x) )
```

We will handle these new predicates as we would handle the original ones.

Furthermore, for every process i, we define the following extended predicate set for PSO:

$$E' := \{\varphi' := \varphi[\texttt{x\_}j\texttt{\_}i/\texttt{x}] \mid \texttt{x} \in \varphi \in E \cup E' \wedge \texttt{x} \in H \wedge j \in [1, s_{x,i}]\},$$

while in the case of TSO, our extended predicate set is defined as follows:

$$E' := \{\varphi' := \varphi[\texttt{buf\_}j\texttt{\_}i/\texttt{x}] \mid \texttt{x} \in \varphi \in E \cup E' \wedge \texttt{x} \in H \wedge j \in [1, s_{x,i}]\}.$$

Though we will handle the latter predicates the same way we would handle the others, unless stated otherwise, we will make note of them being *buffer predicates* and copies of certain different predicates $\in E$.

4.2 The choose statement

The `choose(`*pos,* *neg*`);` statement is defined in subsection 4.3. We will make use of it during our predicate abstraction of assignments. Paraphrased, it states for two boolean variables `A` and `B`:

```
if (A ≡ true)
    choose(A, B) ≡ true
else if (B ≡ true)
    choose(A, B) ≡ false
else
    choose(A, B) ≡ *
endif;
```

Note that boolean variables in Fender may have the value `*`, i.e. indecisive.

4.3 Weakest liberal preconditions

As defined in subsection 4.1 of [2], $WP(\texttt{s}, \varphi)$ is defined as the weakest predicate whose truth before s entails the truth of φ after s terminates (if s terminates), and in the case of s being an assignment statement of the form `x = e`, it equals $\varphi[\texttt{e}/\texttt{x}]$. In this work, we will only compute the weakest liberal preconditions of assignment statements.

4.4 Largest implicative cubes

Paraphrasing the definitions in subsection 4.1 of [2] to better suit the semantics of this work, we define a cube over a boolean variable set V as follows: an implicative cube for a predicate φ is a conjunction of boolean variables $\in V$ (negated or not) representing different predicates $\in E$, which implies φ. We say for a cube C to be the "superset" of another cube D iff D can be expressed as the conjunction of C and a different non-empty cube C' lacking any variable appearing in C. Continuing with our paraphrasing, we define $\mathcal{F}_V(\varphi)$ with the size limit L as the disjunction of all cubes over V implying φ with L or less

literals, and $\mathcal{G}_{\mathcal{V}}(\varphi) := \neg\mathcal{F}_{\mathcal{V}}(\neg\varphi)$. For practicality reasons, and without loss of generality, we can redefine the largest disjunction of implicative cubes as lacking any two cubes of which one is the superset of another. We construct such cube disjunctions conservatively, i.e. once we find that a cube c implies a predicate, we will omit any supersets of c from our disjunction, as described in subsection 5.2 of [2].

4.5   Replacement rules

We use the replacement rules defined in section 4 of [2], albeit in a reduced and slightly adapted form, as we need not consider the caveats of the C language.

4.5.1   Local assignment-, store-, and load statements

We reduce all three types of assignment statements to the same form `l = r`. First, we examine the left-hand side of the assignment. If it is a buffer variable, it will be regarded as the global variable it represents for all purposes with the exception of determining the left-hand sides of all initialization-, assignment- and reset statements. Next, we build the set of relevant predicates R out of any predicate that contains `l`, save for buffer predicates. As long as we encounter predicates containing non-buffer variables, we extend R by all predicates containing the new variables, which will result in the extended set of relevant predicates R'. All the while, we keep ignoring and omitting buffer variables and buffer predicates. Once this closure has been computed, we have all we need to replace the assignment with:

```
begin_atomic;
[initialization of temporary variables]
[assignment of boolean variables]
[reset of temporary variables]
assume(!(𝓕_V(false)));
end_atomic;
```

In this atomic block, we simulate the parallel assignment of the new boolean values to their variables. We define the *initialization of temporary variables* for each predicate $\in R$ with the index j as: `load T_j = B_j;`. Analogously, we define the *reset of temporary variables* as: `T_j = 0;`, which leaves us with the *assignment of boolean variables*, which is also done for each predicate $\in R$ with the index j, and in accordance to subsection 4.3 of [2] as follows:

```
store B_j = choose(𝓕_{R'}(WP(l = r, φ_j)), 𝓕_{R'}(WP(l = r, ¬φ_j)));
```

If the initial left-hand side `l` was a buffer variable, we proceed differently for each store order.

PSO approach We scan through every cube in every disjunction, and we extend the corresponding cube disjunction

$$C \in \{\mathcal{F}_{R'}(WP(\texttt{l = r}, \varphi_j)), \mathcal{F}_{R'}(WP(\texttt{l = r}, \neg\varphi_j))\}$$

with the following cubes:

$$C' := \{c' := c[b'/b] \mid \texttt{l} \in b' \wedge b \mapsto b' \wedge b \in c \in C \cup C'\},$$

where by $b \mapsto b'$, we denote the fact that we can perform repeated replacements of any global variable by one of their buffer representations in a boolean term b, which may be a negated, or non-negated representation of a certain predicate $\varphi \in E \cup E'$, in order to receive an equivalent boolean term b' with the same negation property as b, representing a predicate $\varphi' \in E'$, with all buffer variables in φ and φ' replaced by their global counterparts resulting in the same predicate. To illustrate this relation, consider the following example:

If `B_1` represents the predicate $\varphi_1 := (\texttt{x > y})$, `B_2` represents the predicate $\varphi_1' := (\texttt{x\_1\_1 > y})$, `B_3` represents the predicate $\varphi_1'' := (\texttt{x\_2\_1 > y})$, and `B_4` represents the predicate $\varphi_1''' := (\texttt{x\_1\_1 > y\_1\_1})$, then only the following expressions are correct over these boolean variables: `B_1 ↦ B_2`, `!B_1 ↦ !B_2`, `B_1 ↦ B_3`, `!B_1 ↦ !B_3`, `B_1 ↦ B_4`, `!B_1 ↦ !B_4`, `B_2 ↦ B_4`, `!B_2 ↦ !B_4`.

TSO approach We fetch the global variable represented by $\texttt{l} \equiv \texttt{buf\_}j\texttt{\_}i$ by climbing one step up the abstract semantic tree, which invariably brings us to an if-else statement containing the global variable `x` represented by `l` in the current program point. We have previously illustrated this information deposition by commenting the if-statement. In the resulting pair of cube disjunctions, we first replace every boolean term containing b with

```
choose(b[l/x] && 𝐵(own_j_i = ρ(x)), 𝐵(own_j_i = ρ(x)))
```

$\mathcal{B}(\varphi)$ denotes the boolean variable representing φ. Then, all cubes with boolean expressions containing global variables are replicated once for every possible combination of buffer variables replacing any occurence of a global variable, with the replacements carried out the same way as above. The following example illustrates this round of extension:

Let `choose`($\mathcal{B}$(`buf_1_1 < y`) `&&` $\mathcal{B}$(`own_1_1 = 5`), $\mathcal{B}$(`own_1_1 = 5`)) be one of the replacements made with the above step. Assume $s_i = 3$. Therefore, there are 3 buffer variables, 2 of which might be containing a value of y with $\rho(\text{y}) = 7$. Because of this, 2 replicas of the same cube containing this example term are generated, with this term replaced by the following two terms, respectively:

```
choose(B(buf_1_1 < choose(
  B(buf_1_1 < buf_2_1) && B(own_2_1 = 7), B(own_2_1 = 7)
)) && B(own_1_1 = 5), B(own_1_1 = 5))

choose(B(buf_1_1 < choose(
  B(buf_1_1 < buf_3_1) && B(own_3_1 = 7), B(own_3_1 = 7)
)) && B(own_1_1 = 5), B(own_1_1 = 5))
```

Both replica cubes are added to the cube disjunction.

4.5.2 Conditionals

We consider conditionals of the form `if` (φ) [*if block*] `else` [*else block*] `endif`; and replace them as follows, using the extended relevant predicate set R' of all variables featured in φ, computed the same way as described in the previous subsection:

```
if (*) goto else label;
assume(G_R'(φ));
[if block]
goto end label;
else label: assume(G_R'(¬φ));
[else block]
end label: nop;
```

4.5.3 Assertions

For any predicate φ in the assertion clause that doesn't contain a `pc`(i) counter, we compute its extended relevant predicate set R', as described above, and replace those predicates with $\mathcal{F}_{R'}(\varphi)$.

4.5.4 Overall boolean program structure

We build our final program to be verified with our model checker the following way:

```
shared B_0, ..., B_|E'|-1;
local T_0, ..., T_|E'|-1;

init
    store B_0 = *;
    ...
    store B_|E'|-1 = *;
    load T_0 = B_0;
    ...
    load T_|E'|-1 = B_|E'|-1;

    [predicate abstractions of initialization stores]

    T_0 = 0;
    ...
    T_|E'|-1 = 0;

    assume(!(F_V(false)));

process 1
    assume(!(F_V(false)));
    [predicate abstraction of the process block]
...
[predicate abstraction of the assertion]
```

5 Discussion

We have implemented a prototype of the tool described above. Out of the features we had envised, the alotted time for this thesis allowed us to build a working prototype that can abstract programs running under PSO. In this section, we discuss our expectations of the tool's behaviour and performance. We also show the results of run time measurements made with the tool and we compare it to the implementation of Dan et al. [1].

5.1 Complexity

We examine the time- and space complexities of the individual phases of our computation:

5.1.1 Buffer size analysis

Computing the buffer size increases caused by each program point has a time complexity linear to the number of program points. For PSO programs, computing the actual buffer sizes needed at each program point requires a comparison of all buffer sizes per variable per program point. For TSO programs however, differentiating between variables is not needed. Therefore, the overall time complexity for the buffer size analysis in itself is $\mathcal{O}(mn)$ in PSO, and $\mathcal{O}(m)$ in TSO, with m being the number of program points in P, and n being the number of global variables in P.

Examining the replacement codes of both PSO and TSO, we see that the time we need to generate replacement code blocks is mainly the time of the actual output, as there is no more complex computation necessary to be able to output the code, than the output itself needs. Therefore, generating the replacement codes has the following complexity by statement and store order: store statements: $\mathcal{O}(K)$ in both TSO and PSO; load statements: $\mathcal{O}(K)$ in both TSO and PSO; fence statements: $\mathcal{O}(n)$ in PSO, $\mathcal{O}(K)$ in TSO; flush statements: $\mathcal{O}(nK)$ for TSO and PSO. Therefore, we can conclude that the code replacements following the buffer size analysis are bounded by the $\mathcal{O}(nK)$-complex flush replacement. Since all replaceable statements are program points, the entire process of replacing them is multiplied by m in complexity.

Therefore, the buffer size analysis stage has a time complexity of $\mathcal{O}(mnK)$

5.1.2 Predicate abstraction

Our implementation of the predicate abstraction performed in accordance with our expectation of the Z3 [5] calls being by far the most time-consuming, with run time percentages ranging from an average of 43.5% for Dekker's algorithm, to an average of 76.3% for the alternating bit protocol, as shown in Table 1. This shows that while complexity reduction is always an issue as long as we have problems of an untractable complexity, reducing the number of satisfiability modulo theory (SMT) solver calls has the most potential for speedup.

The complexity of all function calls in the predicate abstraction phase is therefore much less significant than the complexity of the number of SMT calls. Employing the optimisations described in section 4., the number of SMT solver calls is bounded by $|E|$ to the power of the maximum number of terms in a cube. This bound is in practice significantly lower when implicative cubes are found that are below the specified size limit, but this number is highly specific to the algorithm and the corresponding predicates.

5.2 Evaluation

We have implemented the procedures as described so far in a tool we call SRO-TOGAP. Then, we have run numerous tests on the following concurrent al-

gorithms: the alternating bit protocol, Lamport's Bakery Algorithm, Dekker's Algorithm, Peterson's Algorithm, a lock-free queue, and Szymański's Mutual Exclusion Algorithm. All experiments were conducted on an Intel(R) Core(TM) i7-3770 3.40GHz with 8GB RAM. The key questions were whether the buffer size analysis has significantly shortened the time needed to perform a correct predicate abstraction, and whether the replacement techniques employed in the predicate abstraction stage have warded off the complexity explosion that would have resulted from the naïve execution of current predicate analysis methods on the additional predicates spawned from the buffer abstraction.

We compare our results to those of Dan et al. [1], as the measurements had been done on the same algorithms, and the methods were similar. To be more precise: this work seeks to improve on the methods described in [1], and we hope to show that our modifications have helped to improve the performance of those methods. The modifications done are as follows:

Buffer size analysis   As opposed to Dan et al. [1], we perform a buffer size analysis on the program in order to be able to minimise the lines of code generated during our replacement phase by seeking to omit parts which we know never to be reached during run-time. We approach this task by statically analising the input, and performing replacements which can be generated in polynomial time.

Flush abstraction optimisation   In [1], flushes have been modeled in a way that caused each iteration to shift at most K variables, which would have effectively increased the run time of a flush, and the number of program states for each flush operation. In our implementation, the buffer is modelled to be "immobile", i.e. a value stored in the buffer is considered to remain in place until it has been flushed or otherwise deleted. This provides us with the following advantages: each flush iteration only adds one state to the set of program states; the time complexity of flushes is dominated by finding the value to be flushed, not iterating over every value in the buffer; and finally, if the buffer size is exceeded at run time, an immobile buffer tail will always trigger an overflow error at the $K + 1$st consecutive store, while the approach in [1] would non-deterministically cause the buffer tail to recede from the buffer size limit with each flush - though this might be beneficial from an efficient resource allocation point of view, we would like to simplify reaching possible erroneous states for purposes of error detection.

Table 1. Boolean program generation

algorithm	# input predicates	# total predicates	# unique cubes	# SMT calls	time (s)	maximum cube size
Dekker	7	27	15	566	6.01	1
Peterson	7	27	99	572	4.40	2
ABP	8	12	129	488	2.36	2
Szymański	20	70	801	1257	10.05	2
LF Queue	7	32	939	2556	15.24	4
Bakery	15	75	25931	85268	974.71	4

Table 2. Model checking until error state found

algorithm	# states before error	memory used (MB)	evaluation time (s)
Dekker	98	5	0.030
Peterson	98	5	0.034
ABP	327	4	0.045
Szymański	247	7	0.043
LF Queue	157	24	0.388
Bakery	310	36	0.323

All measurements were done under PSO with $K = 5$, i.e. the implementation of the principles discussed for TSO is not included in this work. Furthermore, the model checker has judged all abstractions to be incorrect. This proves that the boolean programs generated by our prototype are not yet verifiable by Fender. Though there was no time to completely debug our code, we can still compare the overall performance of the tool, to get an idea of the potential changes in performance and complexity. Note that we omitted evaluating the ticket locking algorithm, although it had been tested in [1], as our definition of SALPL does not include blocks of atomic code.

Comparing the measurements in [1] with ours, we first note that we have worked the sequential consistency predicates although we have abstracted programs under PSO, as it was our goal to abstract programs using only the predicates needed to prove correctness under sequential consistency constraints. The predicates actually generated by our program are far more numerous and their numbers are less expressive, as their number increases with the multitude of variables and the newly generated predicates do not cause any additional SMT calls.

Another fact we notice is that SROTOGAP generates about 100 times as many cubes as CUPEX [1] for low cube sizes, but less cubes for high cube sizes. This result should be observed with caution however, as in all cases with the cube size limit being > 2, the largest cube disjunction, the disjunction of cubes implying *false*, has timed out in SROTOGAP, which has triggered a halt in that computation, thus preventing the inception of a number of cubes possibly orders of magnitude larger than their counterparts in 13. This comes as no

surprise, as even without SMT calls, the cubes generated using heuristics grows with the number of total predicates $|E \cup E'|$.

The numbers of model checker states are markably lower in our work than they were in [1]. This metric however, along with the other two model checker metrics, is possibly the least significant for our intended comparison, as the model checker has obviously received erroneous outputs from SROTOGAP.

6   Related work

This work is based on the work of Dan, Meshman, Vechev, and Yahav on predicate abstraction for relaxed memory models [1], and it implements most of the techniques derived by the same. It also borrows heavily from the predicate abstraction rules defined by Ball, Majumdar, Millstein, and Rajamani [2]. Below, we describe these publications and the nature of their influence on this work, apart from general inspiration.

6.1   Predicate Abstraction for Relaxed Memory Models

The work by Dan et al. [1] has presented the two abstraction steps this work also employs. It has handled the problem of verifying a program P running on an arbitrary memory model M by embedding a description of said memory model into a program P_M, which is in its behaviour equivalent to P. The resulting abstraction P_M has then been analysed for predicates to be generated in order to be used as correctness criteria for a successive predicate abstraction, which would then describe the conversion of P_M into a boolean program ready to be verified by a model checker. Furthermore, the aforementioned methods have been implemented for TSO and PSO.

It is safe to say that this work primarily seeks to implement the ideas and concepts described in [1], to reproduce its success in achieving feasibility in its implementation, and to improve on its performance and complexity wherever possible.

6.2   Automatic Predicate Abstraction of C Programs

The work by Ball et al. [2] has laid the foundation on which predicate abstraction in this work is implemented. It has described the difficulties model checkers face when confronted with systems with large state spaces, such as software, which is infinite-space. The authors have described the first algorithm to automatically construct a predicate abstraction of programs in C, and they have implemented it in a toolkit, which combined it with other methods such as model checking to statically verify input programs.

The final stage of this work, predicate abstraction, implements the algorithm defined by Ball et al., adjusted for the PSO/TSO language we use, and employs some of the optimizations described by the authors.

7 Conclusion and future work

We were unsuccessful in determining whether this work has improved on the methods in [1]. Though the theories on which this work is built are sound, as proven by [1] and [2], and the expansions thereon are few and simple, the program output is vast, and there can be little information extracted therefrom. This leads us to the first possibility of future development: to write a working implementation of the principles discussed in this work, preferable one that extends SALPL to allow for atomic blocks.

7.1 Future work

Possible further developments in this field are plentiful. We list some of the most apparent aspects that may be improved upon:

TSO implementation Though we did describe the procedure of abstracting programs running under TSO, an efficient implementation would complement the findings of this work and [1].

RMA implementation With some modifications to the parsing, and code abstraction phases of this work, the same analysis method can be adapted for programs running under the Remote Memory Access model, a powerful WMM conceived for buffer-free memory access over the network.

Parallelisation The entire tool, and all its measurements were ran in a sequential manner. While there are some parallelisation chokepoints in the procedure described in this work, e.g. buffer size analysis, code replacement, other, more important parts, such as the computations necessary for the predicate abstraction, are highly parallelisable. With this in mind, the run time of this tool can be further reduced.

7.2 Acknowledgements

References

[1] A. Dan, Y. Meshman, M. Vechev, and E. Yahav. Predicate Abstraction for Relaxed Memory Models. In Static Analysis Symposium '13 (2013)

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In PLDI '01 (2001)

[3] M. Kuperstein, M. Vechev, and E. Yahav. Automatic Inference of Memory Fences. In Formal Methods in Computer Aided Design (2010)

[4] Fender. Model checking, abstract interpretation, and dynamic analysis. ETH Zurich and Technion. http://www.practicalsynthesis.org/fender/

[5] Z3. Theorem prover. Microsoft Research. https://github.com/Z3Prover/z3