Sequentialization Using Timestamps

Anand Yeolekar, Kumar Madhukar, Dipali Bhutada, and R. Venkatesh

Tata Research Development and Design Centre, Pune, India

Abstract. Given a run of a concurrent program and the underlying memory model, we can view the shared memory accesses as a chronological sequence of read and write operations. This chronological sequence of shared memory accesses exactly characterizes the run. We present an approach to sequentialization that captures these sequences by assigning timestamps to the memory accesses. The axioms of the underlying memory model can be encoded as constraints on the timestamps, within the sequentialized program, to generate precisely the set of traces permissible by the original concurrent program. Experimental evaluation shows that the encoding can be efficiently checked by the backend model checker.

1 Introduction

As multi-core processors gain widespread adoption, multi-threaded software is being increasingly designed, developed and deployed. The exponential number of interleavings exhibited by concurrent software poses a challenge during the validation phase of the software development lifecycle. Further, many architectures support weak memory models allowing concurrent program behaviours that need not conform to sequential consistency. Consequently, model checking is necessary to exhaustively check the concurrent program for *heisenbugs* - bugs that lie deep inside an interleaving and are near-impossible to detect or reproduce using testing. In this work, we present an approach to sequentialization of concurrent C programs to enable efficient checking of program assertions.

Consider two threads T_1`:x++;` and T_2`:x--;` , incrementing and decrementing a shared variable, respectively. The threads issue a read operation of the shared variable from memory and then a write to store the updated value. Let r^i and w^i denote the read and write operation, resp., for thread i. Under Sequential Consistency (SC) memory model [1], any run must satisfy the following conditions: (i) program order be maintained among operations within a thread, and (ii) the execution appears to be the result of a single sequential order across threads (atomicity). The set of sequences of the shared memory read r and write w operations, corresponding to all possible runs of the threads, can be listed as: $\{\langle r^1, r^2, w^1, w^2\rangle, \langle r^1, r^2, w^2, w^1\rangle, \langle r^1, w^1, r^2, w^2\rangle, \langle r^2, w^2, r^1, w^1\rangle\}$. Note that we ignore sequences that differ only in a read sub-sequence permutation, for e.g. $\langle r^2, r^1, w^1, w^2\rangle$.

It is well-known [3, 16] that the set of all *correct* sequences can be captured as solutions to constraints derived from the program and the underlying memory model. For the example shown above, we obtain two relations from the threads, namely $po(r^1, w^1)$ and $po(r^2, w^2)$, where po denotes the per-thread program order between memory accesses. Additionally, as per condition (i) of SC stated earlier, $po(m_1, m_2) \Leftrightarrow hb(m_1, m_2)$ must hold for all m_1, m_2 belonging to the same thread, in every correct sequence. We say that $hb(m_1, m_2)$ holds in a sequence if m_1 precedes m_2 in that sequence. Thus we obtain the constraint $hb(r^1, w^1) \wedge hb(r^2, w^2)$, which can be solved to get precisely the sequences listed above.

The hb relation is commonly referred to as the *happens before* relation in the literature, relating memory accesses in an execution trace. The second condition of SC specifies how a given sequence or trace can be interpreted: a read must return the value of the *freshest* write, i.e. if a read r links to a write w then $\nexists w' :$ $hb(w, w') \wedge hb(w', r)$, unless w, w' write to different memory locations. Observe that both the SC conditions can be expressed using the happens-before relation, which can be naturally modeled if we could refer to the *time* of occurrence of the shared memory operations. For example, if t_m and $t_{m'}$ denote the time of occurrence of memory access m and m' respectively, then $hb(m, m')$ is simply the constraint $t_m < t_{m'}$.

In this paper, we propose the use of *timestamps* for sequentialization. A timestamp is a natural number that encodes the logical time of occurrence of a shared memory access. We assign timestamps to shared memory accesses to map reads with writes as permitted by the underlying memory model. The set of permitted read-write maps is defined by the axiomatic specification of the memory model. It is this specification that we encode as constraints on timestamps, at the source level.

We construct a sequentialized program, that encodes the constraints on timestamps described above, as follows. We introduce global arrays to store timestamps of writes along with their values. Timestamps are assigned non-deterministically and constrained to be monotonically increasing. We encode the requirements for SC by rewriting instructions accessing shared memory i.e., read and write operations. The program is instrumented as follows: (i) a *write* access is redirected to an array location whose timestamp is larger than a *locally* tracked current time, (ii) a *read* reads from a location with a timestamp *closest* to the current time i.e. the timestamp of the *successive* write (in the array) must be larger than current time. The sequentialization is completed by issuing calls to the thread function bodies directly. Locks are modeled as shared variables. Locking sets the variable provided the latest access to the variable was a reset, and unlocking resets the variable. We show in Section 2.3 that the sequentialized program exhibits precisely the set of behaviours of the concurrent programs.

We propose that using timestamps can *naturally* describe the runs of a concurrent program under any memory consistency model and yield a simple yet efficient sequentialized program. We make the following contributions through this work.

– A sequentialization approach based on timestamps that encodes axioms of SC.
– A prototype tool, ConSequence, that implements the proposed encoding.

– An experimental evaluation demonstrating the usability of this approach.

The rest of the paper is organized as follows. In Section 2, we illustrate our encoding under SC with an example, formalize the encoding and present an argument for its correctness. Our experimental results are presented in Section 3. We discuss the related work in Section 4 before concluding in Section 5 and listing some immediate directions of future work.

2 Sequential consistency memory model

2.1 Illustrative example

```
#include <pthread.h>
#define N 3
int i=1, j=1;
void* f1(void* arg){
 int x;
 for(x=0;x<N;x++) {
  i = i+j; }
}
void* f2(void* arg){
 int x;
 for(x=0;x<N;x++) {
  j = j+i; }
}
int main() {
 pthread_t t1,t2;
 pthread_create(&t1,0,f1,0);
 pthread_create(&t2,0,f2,0);
 pthread_join(t1,0);
 pthread_join(t2,0);
 assert(i<21);
}
         (a)
```

```
#define N 3
int i=1, j=1;

f1(){
 int x;
 for(x=0;x<N;x++) {
  write_i(read_i()+read_j());}
}
f2(){
 int x;
 for(x=0;x<N;x++) {
  write_j(read_j()+read_i());}
}
int main() {
 sysinit();
 procinit(); t1(); procend();
 procinit(); t2(); procend();
 sysend();

 assert(i<21);
}
           (b)
```

Fig. 1. Example concurrent program `fib.c` (a) and its sequentialization (b)

We illustrate our approach with an example, shown in Figure 1(a), that computes the Fibonacci sequence with two threads, under SC. The assertion can be violated only when context switches between the threads follow a certain order (the reads and writes occur hand-in-hand). This makes the analysis challenging for tools that rely on under-approximations such as write-bounding or context-bounding.

The sequentialized code is shown in Figure 1(b). We use two procedures, `write_var()` and `read_var()`, for every shared variable `var`, to instrument its memory writes and reads, respectively. Thread creation in `main` function is replaced with direct calls to the thread function body, augmented with pre- and post-processing code.

```
#define MAXW_i (N+1)
#define MAXW_j (N+1)
#define MAXT ((MAXW_i-1)+(MAXW_j-1)+1)

int *value_i,*value_j;
unsigned short
     *ts_i,loc_i=0,count_i=0,last_i=0,
     *ts_j,loc_j=0,count_j=0,last_j=0,
     ct=0;
_Bool *free_i,*free_j;

void sysinit(){
 value_i=(int *)malloc(
          sizeof(int) * MAXW_i);
 value_i[0]=i;

 ts_i=(unsigned short*)malloc(
       sizeof(unsigned short)*(MAXW_i+1));
 ts_i[0]=0; ts_i[MAXW_i]=MAXT;
 for (int k=1; k<MAXW_i+1; k++)
   assume(ts_i[k-1] < ts_i[k]);

 free_i=(_Bool *)malloc(
         sizeof(_Bool)*MAXW_i);
 free_i[0]=false;
 /* similarly for j */ }

void procinit() {
 ct=0; loc_i=0; loc_j=0;}
```

```
void procend() {
 if (last_i<loc_i) last_i=loc_i;
 if (last_j<loc_j) last_j=loc_j;}

void sysend() {
 assume(last_i==count_i);
 i=value_i[last_i];
 /* similarly for j */ }

void write_i(int value) {
 unsigned short loc=*;
 assume(loc_i < loc < MAXW_i &&
        free_i[loc] &&
        ts_i[loc] > ct &&
        value_i[loc] == value);
 loc_i = loc;
 free_i[loc] = false;
 icount++;
 ct = ts_i[loc];}

int read_i() {
 unsigned short loc=*;
 assume(loc_i <= loc < MAXW_i &&
        ts_i[loc+1] > ct);
 loc_i = loc;
 if (ct<ts_i[loc]) ct=ts_i[loc];
 return value_i[loc];}

  /* similarly for write_j, read_j */
```

Fig. 2. Datastructures and auxiliary code for sequentialization

The auxiliary datastructures and code used for sequentialization is shown in Figure 2. We assume `fib.c` is *structurally bounded* with loops executing `N` number of times. Let `MAXW_var` denote the maximum number of writes to shared memory variable `var` that may occur along *any* program path, across all threads of the program. In the example of Fig. 1(a), `MAXW_i = MAXW_j = N+1`, as each thread writes to a variable exactly once in a loop iteration, apart from the initialization. Additionally, we define `MAXT` as the total number of unique timestamps needed for write accesses across all shared variables, where initializations of all shared variables get the same timestamp.

For each shared variable, we use additional memory as explained here. Arrays `value_var,ts_var` store the value of a write access and its timestamp, respectively, and `free_var` tracks if an index in value and timestamp arrays is *available*. An index ceases to be available once it is written to. We refer to the three arrays as a *timestore* for the shared variable. Auxiliary variable `count_var` records the number of writes and `last_var` tracks the largest index accessed by a read or write in each timestore. The variable `ct`, common to all shared memory variables of the concurrent program, tracks the time of the latest memory access issued by a thread procedure. Note that `ct` is updated locally by each thread, though declared as a global variable.

Procedure `sysinit()` initializes each timestore to non-deterministic values (timestamps are bounded by the respective maximum number of writes along any path) through `malloc` calls. It further adds the constraint that timestamps in `ts_var` increase strictly monotonically.

We explain the write and read instrumentation scheme wrt shared variable `i` of Fig. 1(a), presented in the procedures `write_i()` and `read_i()` of Fig. 2. Intuitively, in the sequentialized program, a write advances the *local* clock to allow for interfering writes from other threads to happen. We select an empty location by non-deterministically advancing from the current location `loc_i` in the timestamp array and store the value (`value_i[loc]==value`). We also add a constraint to ensure that this write occurs *after* the current time `ct` (`ts_i[loc]>ct`). A read in the sequentialized program, may return the value at any index of the timestore, provided this is the most *recent* write relative to the read. We first select a write in the timestore array by non-deterministically advancing from the last accessed (read or write) location by this thread. Next, to ensure that this is the most *recent* write relative to the read, we add a constraint that the *successive* write occurs after the current time (`ts_i[loc+1]>ct`), which is intuitively the time at which the read happens. Recall that the timestores are sorted on timestamps *a-prióri*; this guarantees the succession of writes. Note that an explicit assignment of timestamps to read accesses is not required to encode the aforementioned constraint; we thus do not assign timestamps to read accesses.

The procedure `procinit()` resets variables `ct` and `loc_i, loc_j`. The procedure `procend()` tracks the last write location updated by thread procedures. Finally, procedure `sysend()` ensures that writes are stored contiguously in the timestore by adding the constraint `last==count` for each shared variable and finally reinstates the shared variables.

	`value_i[]`	`ts_i[]`	`free_i[]`
0	1	0	false
1	3	2	false
2	8	4	false
3	21	6	false
4	-	7	-

	`value_j[]`	`ts_j[]`	`free_j[]`
0	1	0	false
1	2	1	false
2	5	3	false
3	13	5	false
4	-	7	-

Fig. 3. Datastructures populated by the counterexample produced by Cbmc

The resulting sequentialized program can be analyzed by any sequential model checker such as Cbmc [6]. Figure 3 shows how the datastructures are populated by the counterexample returned by Cbmc, violating the assertion, for `N=3`.

2.2 Formalization

Let P_C be a *structurally bounded* concurrent C program consisting of threads $T_1, .., T_n$, invoking procedures $f_1, .., f_n$, respectively, using the `pthreads` API. Let V denote the set of variables shared by the threads. We assume that procedure `main` invokes the threads and waits for the threads to `join`, followed by an assertion ϕ to be checked. Let G^k denote the unfolded control flow graph of thread id k, with each statement containing at most one read r or write w access to a shared variable. We denote a memory access by m when we do not distinguish between a read or write. We use the notation m^v to represent the memory access m operates on the shared variable v.

Definition 1. *The per-thread program order po is a relation that statically orders memory accesses.*

$$\forall m, m', path(m, m') \Leftrightarrow (m, m') \in po \tag{1}$$

where $path(i, j)$ holds iff there is a path from i to j in G^k.

We encode *po* in terms of *happens-before* relation $\hat{hb}$, i.e hb (stated in Sec. 1) restricted to same-thread memory accesses, as follows.

$$\forall m, m', po(m, m') \Leftrightarrow \hat{hb}(m, m') \tag{2}$$

Any interleaving or trace of P_C is a sequence τ of memory accesses that is a solution to the *po* constraints encoded as the $\hat{hb}$ relation (Eqn. 2). The interpretation of τ, in terms of the values of the memory accesses, comes from the underlying memory model as a *read-from* relation.

Definition 2. *The read-from relation rf maps every read to a write in τ.*

Under SC, rf enforces the condition that in a trace, a read returns the value of the *most recent* write to the same variable. We refer to this condition as *atomicity.*

We encode rf in terms of the happens-before relation:

$$\forall r \in \tau, \exists w \mid val(r) = val(w) \wedge hb(w, r) \wedge \nexists w' : hb(w, w') \wedge hb(w', r) \tag{3}$$

where $val(.)$ returns the value of the memory access and we interpret $hb(i, j)$ over the trace as i precedes j in τ.

Timestamps allow us to model the hb relation naturally and succinctly. In fact,

$$\forall m, m', hb(m, m') \Leftrightarrow t_m < t_{m'} \tag{4}$$

Sequentialization The sequentialization is presented in example Fig. 1(b) and 2. The listing in Fig. 2 encodes the SC conditions stated in Eqn. 2 and 3, in procedures `read` and `write`.

In the next subsection, we show that the encoding correctly captures these conditions in terms of timestamps (Eqn. 4).

Note that in Eqn. 4, the inequality need not be strict when m' is a read access. It suffices to have distinct timestamps for writes. As an optimization, wherever possible, we allow a read access to implicitly acquire the timestamp of the preceding memory access, whether read or write.

2.3 Correctness of the sequentialization

In this section we take a closer look at the auxiliary code for sequentialization shown in Figure 2. Recall that the timestore records writes from all threads in a single global order, obtained by sorting the timestamps a priori. Also, procedures `read` and `write` are the ones that encode the axioms of the underlying memory model, when accessing timestore.

The intra-thread program order, encoded as constraints on timestamps, is preserved through (i) the constraint `ts[loc]>ct` when writing and `ct` getting updated to `ts[loc]`, and (ii) advancing `ct` when reading in the future.

Lemma 1. *The conditions listed above guarantee program order between per-thread memory accesses.*

Proof. Consider $m, m' \mid (m, m') \in po$. When m' is a write access, then condition (i) ensures that $t_m < t_{m'}$ as a write always updates `ct` to a larger value. Note that `ct`, intuitively the current time, is at least as large as t_m when the memory access m has occurred. When m' is a read, the timestamp of m' is either the same as `ct` (when reading in the past) or advanced (when reading in the future) as per condition (ii). This ensures $t_m <= t_{m'}$ (note that the equality on timestamps is an optimization not violating SC, as described earlier). Thus the proposed sequentialization guarantees the per-thread program order.

The atomicity condition of SC is preserved by (i) the constraints `iloc<loc` and `iloc<=loc` when writing and reading, respectively, and (ii) the constraint `ts_i[loc+1]>ct` when reading.

Lemma 2. *The conditions (i) and (ii), above, guarantee atomicity under SC.*

Proof. The variable `ct` also accounts for thread interference. Whenever a read maps to a write of a different thread, the timestamps of the read and write are compared. If the write has occurred in the past relative to the read, we ensure the atomicity condition of SC by constraining the *successive write in global order* to occur *after* this read, through the constraint `ts_i[loc+1]>ct`. If the write is located in the timestore in the future wrt. this read, then this implies that the read should have occurred *after* this write and *before* the subsequent write (in global order); thus the current time, which is the time this read happens, is updated to match the write's time, again guaranteeing atomicity.

The existential quantifier stated in the SC atomicity condition is handled implicitly in our implementation by sorting the timestore during the initialization. The timestore reflects the write serialization to main memory i.e. the sequence of writes to main memory as agreed upon by all threads.

To summarize, shared memory read and write operations are constrained exactly according to the axioms of the memory model. The runs of the sequentialized program can be obtained by solving the read and write constraints in addition to the program logic. This set of runs exactly corresponds to the set of runs of the parallel program, since the runs of the sequentialized program are exactly the solutions to the system of constraints of the axiomatic memory model, encoded at source level.

Since our sequentialization approach does not impose (or relax) any other constraints, the only constraints present in the sequentialized program are the memory model axioms which define the correct behavior. Therefore, the transformation precisely captures correct behavior. The approach easily generalizes to a framework where one can plug the memory model axioms, akin to a library call, to obtain a transformation corresponding to a different memory model.

Corollary 1. *The proposed encoding correctly captures the constraints of SC at the source level i.e. the sequentialization exactly encodes the behaviours of the the concurrent program.*

3 Experiments

We have implemented the timestamp-based sequentialization approach in a prototype tool ConSequence. Given a structural or unwind bound u, ConSequence transforms the multi-threaded C code by instantiating a header file from a template, which can be viewed as a library that replaces thread API calls. Read and write accesses to shared memory variables are identified using PRISM[1]and replaced with calls to respective procedures. ConSequence automatically computes the maximum number of writes for every shared variable. Thread creation and joining calls are replaced with calls to thread procedures with pre and post processing code. ConSequence uses CBMC to check assertions in the resulting sequentialized program (other model checkers can be used). For efficient performance with CBMC, ConSequence implements several tweaks/optimizations during the sequentialization.

Table 1 shows the comparison between ConSequence, MU-CSeq 0.3 [18], CBMC 5.5 [6], Corral [2] and CIVL [15]. Though MU-CSeq 0.3 comes with CBMC 4.9 by default, in order to have an unbiased evaluation, we have run both ConSequence and MU-CSeq with CBMC 5.5 as the backend model checking tool. The benchmarks have been selected from the concurrency category of verification tasks at SV-COMP 2016 [4], except the ones numbered 4 and 10 that are synthetic. `inc-dec` has 2 threads and one shared variable incremented and decremented under a lock, in a loop. `3var` has 3 shared variables with two threads

[1] A static analysis framework developed at TRDDC, Pune [11, 5].

Table 1. Results for Sequential Consistency

S. No.	Benchmark	Unwind	Assertion	ConSequence	MU-CSeq 0.3	Cbmc 5.5	Corral	CIVL
1.	`stateful01.c`	1	safe	0.17	0.53	1.18	53.71	1.72
2.	`stateful01.c`	1	unsafe	0.17	0.52	1.07	47.64	1.72
3.	`fib_bench_longer.c`	7	safe	2.8	17.12	17	timeout	timeout
4.	`3Var-nolock-2threads.c`	4	safe	0.65	38	7.21	16.5	18.4
5.	`27_Boop.c`	10	unsafe	1.1	52.7	2.75	`error`	timeout
6.	`fib_bench_longest.c`	12	safe	130	timeout	timeout	timeout	timeout
7.	`fib_bench_longest.c`	12	unsafe	101	543.6	749	`error`	timeout
8.	`peterson.c`	60	safe	1.5	42.9	15.3	2.07	1.61
9.	`szymanski.c`	21	safe	2.30	45.02	7.5	1.91	1.65
10.	`inc-dec-lock-2threads.c`	9	safe	8.9	44.5	timeout	timeout	4.8
11.	`dekker.c`	9	safe	6	37	4.94	`error`	timeout

incrementing and decrementing the variables in a loop. For every benchmark, we report the unwinding depth chosen, state whether the benchmark was safe or unsafe for the chosen unwinding, and list the time taken (in seconds) by the tools to analyze the benchmark correctly. We set a timeout of 60 seconds for these experiments, except for `fib_bench_longest` program (rows 6 and 7) where we set a timeout of 900 seconds. All the benchmarks, except `fib_bench_longest`, were run with the largest possible unwind such that at least 3 out of 5 tools run to completion, i.e. we allowed no more than two tools to time out. The term `error` indicates that the tool either crashed, or terminated without producing the correct result. For the comparison to be unambiguous, the reported values of time are in fact the time taken by the underlying decision procedure in the analysis, for all the tools except CIVL. In case of CIVL we could not determine the time taken by the decision procedure separately, and hence we have reported the total time taken. We conducted our experiments on an Intel Xeon 2.2 GHz 32-core machine with 20 GB RAM. As seen from Table 1, ConSequence outperforms all other tools on both safe and unsafe instances. The benchmark programs, their sequentialized version (as produced by ConSequence), the exact commands used to invoke the tools, and the corresponding log files are available at `http://www.cmi.ac.in/~madhukar/ConSequence/`.

The graphs shown in Fig. 4 compare the number of variables and clauses generated by ConSequence and MU-CSeq (by the backend decision procedure of Cbmc, in both the cases), during the analysis of benchmarks shown in Table 1. Our encoding consistently generates fewer variables and clauses, explaining the order of magnitude reduction in the model checking time with ConSequence.

4 Related work

The idea of sequentialization was proposed by Qadeer and Wu [14] with the motivation to leverage analysis techniques developed for sequential programs. Lal and Reps [13] proposed a sequentialization for a given bound on context switches. Their scheme implemented a non-deterministic scheduler by instrumenting the code and storing the program state at each switch. An improved version of this algorithm was implemented in [7]. Inverso et al. proposed a further enhancement

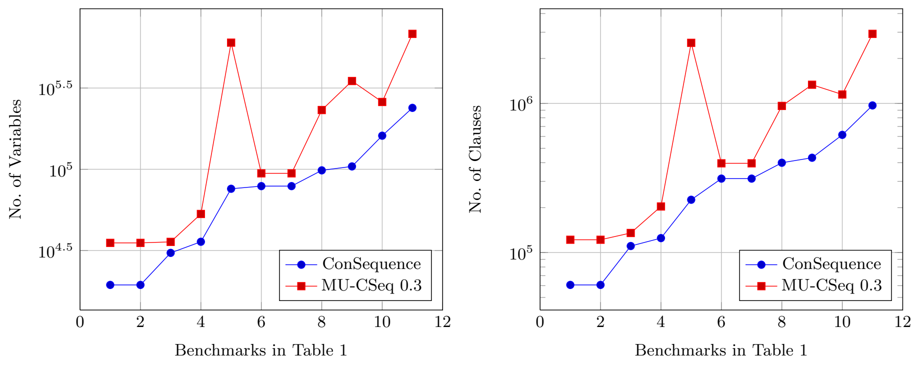

Fig. 4. Plots comparing the number of variables (left), and the number of clauses (right), as reported by the backend decision procedure in ConSequence and MU-CSeq 0.3, for the benchmarks shown in Table 1.

[8, 9] combining [12] and bounded model checking. Recently, Tomasco et.al. [19] proposed a new technique for sequentialization that bounds the number of shared memory write accesses. This approach explores an orthogonal set of interleavings compared to the earlier context-bounding approaches.

Tomasco et.al. [19] propose memory unwinding, i.e. a sequence of write operations into the shared memory. The technique guesses the sequence and simulates the executions of a multi-threaded program according to any scheduling that respects it. However, it bounds the total number of write operations into the shared memory. To track writes, they use arrays and duplicate the shared memory state (of unmodified variables) each time a write occurs in a thread (read-explicit scheme), or use pointers to track the last relevant write at each memory location (read-implicit scheme). The main difference is how time is stored and used to form constraints: in [19], the array location implicitly represents time whereas we use an auxiliary array to explicitly store timestamps. The advantage of our encoding is that the successive write is actually stored in the next location. This avoids costly duplication of the read-explicit scheme and maintaining pointers in the read-implicit scheme, scaling better as the number of memory accesses and/or threads increases. Further, linking constraints can be naturally and compactly expressed with respect to timestamps instead of locations of (arrays representing) shared memory.

The memory unwinding technique has been extended further [20] to use timestamps and for the analysis of TSO and PSO [21]. The authors view a concurrent program as two independent subsystems, separating computation (individual threads) from communication (shared memory). This is similar in spirit to our approach but we differ in the encoding and implementation details. This reflects as an order of magnitude reduction in both the number of clauses and the number of variables for the benchmarks (see Fig. 4).

Partial orders employing memory model axioms to link read and write events are presented in [16, 3, 10]. A two-stage approach involving intra-thread summarization and composition under sequential consistency is presented in [16, 17]. The method constructs a concurrent control flow graph (as part of an *interference skeleton*) to discover intra-thread causal ordering of events. The linking phase gives rise to redundant pairing of *read-write* events, requiring pruning using dataflow analysis. In contrast, our approach works at source-level, provides a syntactic transformation under SC, and does not require any pruning of constraints. Several sequential program analysis techniques can be applied on the transformed program including procedure summarization.

5 Conclusion and future work

We have presented an approach to sequentialization of concurrent programs that uses timestamps to map reads with appropriate writes. The possible read-write maps, defined by the axiomatic memory model, are encoded as constraints over the timestamps. The solutions to these constraints yield traces that precisely capture all valid interleavings of the concurrent program. Our encoding, based on timestamps, naturally captures the semantics of memory models expressed as axiomatic composition rules on reads and writes. Further, the encoding is compact, simple and efficiently analyzable by a bounded model checker like CBMC.

The use of model checkers to explore interleavings encoded as constraints on timestamps has the potential to face state space explosion problem. In particular, the technique may explore redundant orders of writes whenever the number of writes produced some choice of paths is less than the (statically) allocated space in the timestore. Further, when the size of shared memory is large, such as when entire arrays are shared, the timestores may become prohibitively large to analyze. Another limitation is that the programs need to be structurally bounded, which effectively bounds the number of threads by bounding loops and recursive function calls. However, given that bounded model checking is the most popular form of model checking used in the industry, this limitation may be acceptable in practice.

The future steps for this work are:

- Improve tool support for more thread operations from posix-API.
- Optimize the timestamps space by assigning the same timestamps to independent writes (or fixing their order of exploration). Note that the size of timestamps directly affects the model checker's state space.
- Use invariants to reduce the timestore size.
- Extend the work to support relaxed memory models i.e. program order and atomicity relaxation.

References

1. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29, 66–76 (1995)

2. Akash Lal, Shaz Qadeer, S.L.: Corral: A solver for reachability modulo theories. Tech. rep. (January 2012), `https://www.microsoft.com/en-us/research/publication/corral-a-solver-for-reachability-modulo-theories/`
3. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Computer Aided Verification. pp. 141–157. Springer (2013)
4. Beyer, D.: Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In: Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. pp. 887–904 (2016)
5. Chimdyalwar, B., Kumar, S.: Effective false positive filtering for evolving software. In: Proceedings of the 4th India Software Engineering Conference. pp. 103–106. ISEC '11, ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/1953355.1953369`
6. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 2988, pp. 168–176. Springer (2004)
7. Fischer, B., Inverso, O., Parlato, G.: Cseq: A concurrency pre-processor for sequential c verification tools. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. pp. 710–713. IEEE (2013)
8. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded model checking of multi-threaded c programs via lazy sequentialization (2014)
9. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Lazy-cseq: A lazy sequentialization tool for c. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 398–401. Springer (2014)
10. Kahlon, V., Gupta, A., Sinha, N.: Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In: Computer Aided Verification. pp. 286–299. Springer (2006)
11. Khare, S., Saraswat, S., Kumar, S.: Static program analysis of large embedded code base: An experience. In: Proceedings of the 4th India Software Engineering Conference. pp. 99–102. ISEC '11, ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/1953355.1953368`
12. La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: Computer Aided Verification. pp. 477–492. Springer (2009)
13. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design 35(1), 73–97 (2009)
14. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: ACM SIGPLAN Notices. vol. 39, pp. 14–24. ACM (2004)
15. Siegel, S.F., Dwyer, M.B., Gopalakrishnan, G., Luo, Z., Rakamaric, Z., Thakur, R., Zheng, M., Zirkel, T.K.: Civl: The concurrency intermediate verification language. Tech. Rep. UD-CIS-2014/001, Department of Computer and Information Sciences, University of Delaware (2014)
16. Sinha, N., Wang, C.: Staged concurrent program analysis. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. pp. 47–56. ACM (2010)
17. Sinha, N., Wang, C.: On interference abstractions. In: ACM SIGPLAN Notices. vol. 46, pp. 423–434. ACM (2011)

18. Tomasco, E., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: MU-CSeq: Sequentialization of C Programs by Shared Memory Unwindings, pp. 402–404. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
19. Tomasco, E., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Verifying concurrent programs by memory unwinding. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 9035, pp. 551–565. Springer Berlin Heidelberg (2015)
20. Tomasco, E., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Verifying concurrent programs by memory unwinding. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 551–565. Springer (2015)
21. Tomasco, E., Nguyen Lam, T., Fischer, B., La Torre, S., Parlato, G.: Separating computation from communication: A design approach for concurrent program verification (2016)