LLMC: Verifying High-Performance Software

Freark I. van der Berg(✉)

Formal Methods and Tools, University of Twente,
Enschede, The Netherlands
f.i.vanderberg@utwente.nl

Abstract. Multi-threaded unit tests for high-performance thread-safe data structures typically do not test all behaviour, because only a single scheduling of threads is witnessed per invocation of the unit tests. Model checking such unit tests allows to verify all interleavings of threads. These tests could be written in or compiled to LLVM IR. Existing LLVM IR model checkers like DIVINE and Nidhugg, use an LLVM IR interpreter to determine the next state. This paper introduces LLMC, a multi-core explicit-state model checker of multi-threaded LLVM IR that translates LLVM IR to LLVM IR that is *executed* instead of interpreted. A test suite of 24 tests, stressing data structures, shows that on average LLMC clearly outperforms the state-of-the-art tools DIVINE and Nidhugg.

1 Introduction

High-performance software often uses thread-safe data structures to allow multiple threads access to the data, without corrupting it. Unit tests for such data structures typically do not test all behaviour, because the thread scheduler of the run-time environment non-deterministically chooses only a single interleaving. Thus, only a single trace is witnessed each time the unit test is invoked. If we would *model check* [1] these unit tests, we can witness all possible traces by exploring all thread schedules. Because it does not depend on the run-time environment, model checking can become part of a continuous integration pipeline, enabling push-button verification of multi-threaded software.

These thread-safe data structures can be written in or compiled to LLVM IR, the intermediate representation of the LLVM Project [2]. The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Many front-ends for LLVM IR exist, for example for C, C++, Java, Ruby, and Rust, potentially allowing an LLVM IR model checker to be usable for many languages.

1.1 Related Work

Model checkers that operate on LLVM IR already exist, for example DIVINE, Nidhugg, RCMC and LLBMC. DIVINE [3] is a stateful multi-core model checker of multi-threaded LLVM IR. It has many features such as capturing I/O during model checking, SC and TSO memory models, library support such as `libc` and `libpthread`. Input programs are linked with DIVINE's operating system layer, DiOS, and are interpreted as a whole on the DiVM virtual machine.

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 690–703, 2021.
https://doi.org/10.1007/978-3-030-81688-9_32

DIVINE detects memory operations to thread-private memory, by traversing the heap on-the-fly and recognizing if a memory-object is either known only to one thread or to multiple [4]. In the former case, memory operations to that memory-object can be *collapsed*, i.e. joined with the previous instruction.

Nidhugg [5] is a stateless multi-core model checker of multi-threaded LLVM IR that uses an LLVM IR interpreter. It features a sophisticated partial-order reduction, *rfsc* [6], that categorizes traces according to which read reads from which write and traverses only one trace in each category. In practice this reduction is quite powerful. However, Nidhugg comes with a caveat: because Nidhugg is stateless, common prefixes of traces are traversed once per trace instead of once in total. This down-side of a stateless approach becomes more pronounced with longer and more often occurring common traces. Moreover, Nidhugg might not terminate in the presence of infinite loops.

RCMC [7] is also a stateless LLVM IR model checker. During execution within its LLVM IR interpreter, it keeps track of a happens-before graph of all observed memory operations. Using this, RCMC can determine the possible values a read can observe, without simply executing all interleavings of all threads. Unlike Nidhugg, it does not support heap memory and is only released in binary form.

CBMC [8] is a bounded model checker for C and C++ programs, using SMT solving to check for memory safety, exceptions, undefined behaviour and assertions. Loops and recursion are a problem for CBMC when their bound cannot be determined: one needs to set an upper bound on the number of unwindings.

LLBMC [9] is similar to CBMC, using SMT-solving to find bugs, but only for single-threaded C/C++ programs and it operates on LLVM IR.

Other, less related tools include SMACK [10], SeaHorn [11] and KLEE [12].

1.2 Contribution

This paper introduces LLMC 0.2, a stateful multi-core model checker of multi-threaded LLVM IR. Instead of using an LLVM IR interpreter like DIVINE, Nidhugg and LLMC 0.1 [13], it transforms input LLVM IR to LLVM IR that implements the DMC API, the next-state interface to the model checker DMC [14]. We call this transformation process LL2DMC and combined with DMC (Fig. 1), it allows for up to three orders of magnitude higher throughput (states/s) than DIVINE. At present, LLMC lacks sophisticated state space reductions, causing state space sizes of roughly two orders of magnitude larger than DIVINE. We compared LLMC to DIVINE and Nidhugg using a test suite covering various data structures. Overall, despite the lack of sophisticated reductions, LLMC is on average an order of magnitude faster than DIVINE and ∼3.8x faster than Nidhugg. Additionally, LLMC is able to compute the state spaces of the tests where DIVINE or Nidhugg fail.

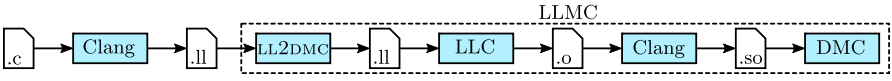

Fig. 1. The flow of how an LLVM IR input program is verified in LLMC.

2 LLMC: Low-Level Model Checker

This section explains how the transformation process (LL2DMC) transforms the input LLVM IR of a program to LLVM IR that implements the DMC API. LLMC supports LLVM IR compiled from C and C++, by handling a number of builtins (e.g. `__atomic_*` for atomic memory operations), part of `libpthread` (for thread support), `libc` (e.g. for memory allocation) and global constructors.

2.1 DMC Model Checker

The model created by LL2DMC is given to DMC to explore. DMC interacts with the model via the DMC API (NEXTSTATE API and DTREE API combined) as illustrated in Fig. 2: after requesting the initial state from the model, DMC continues to request successor states, until the state space has been generated. A state is a vector of 32-bit integers; two states need not be of the same length.

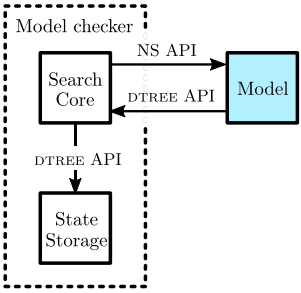

Fig. 2. DMC model checker

The states are stored in the concurrent compression tree DTREE [14], allowing lossless compression, fast insertion and duplicate detection of states. When inserted, states are given a unique `StateID`. A `StateID` can be stored in states as well, thus allowing the creation of a DAG of states: a *root-state* and *sub-states*. Additionally, DTREE allows incremental updates to a state, without having the actual contents of the state and it allows partial reconstruction of states. This *delta interface* uses the `StateID` to identify states and can avoid needless copying of entire states, increasing performance. DMC exposes these DTREE features as part of the DMC API [14].

2.2 Input Language to LL2DMC: LLVM IR

To understand how LLMC handles input LLVM IR [2], we briefly explain it here. LLVM IR supports control flow by way of basic blocks. Basic blocks are a list of instructions that execute sequentially. The last instruction of a basic block is a terminator instruction, such as a branch (jump) instruction or `return` statement.

LLVM IR uses single static assignment form for register values. To support data flow depending on control flow, ϕ-nodes exist. These nodes are instructions at the beginning of a basic block that take a value depending on the basic block from which was jumped to the basic block containing the ϕ-nodes.

2.3 Output of LL2DMC: Model Implementing DMC API

The output of LL2DMC is a model that implements the NEXTSTATE API part of the DMC API of the model checker DMC [14]. The NEXTSTATE API requires two interfaces from a model: one to communicate the initial state and one to generate next states, given a state.

The *initial state* of a model generated by LL2DMC is as if one just started the program: registers are unused, global memory is initialized to 0 and a call to the global constructor (`@llvm.global_ctors`) is set up. Global constructors are functions that are called before `main`, which are used to initialize memory and miscellaneous initialization, such that the executable is set up properly before `main` is invoked. Having the initial state in this manner, allows the global constructor to be part of the state space and thus be checked as well.

Starting with the initial state, DMC will keep asking the model to generate the next states for a given state, by invoking the *next-state interface* of the model, until there are no more new states of which to request next states. Given a state, the next-state interface determines the states reachable from that state. In the case of a model generated by LL2DMC, first the global constructors of the modelled program are explored, thus faults in global constructors are detected. When the global constructors are completed, a call to `main` is set up. At this point, the exploration is performed until no new states are visited.

2.4 State Space Exploration

This section describes the next-state function and how it is generated from LLVM IR. Figure 3 describes what a state looks like. A state contains information not unlike what an operating system keeps track of [15]. All instructions are mapped to a unique index, such that the `PC` (program counter) uniquely identifies the current position in code. The field `Thread Results` holds the return values of finished threads; the field `#threads` specifies the number of threads in the current state. The remainder of the state constitutes a list of per-thread data.

Each thread has its own `PC` and can independently manipulate it by function calls or branching. `Status` fields are used to indicate whether the thread/program is running, done or failed. Each thread has its own set of `Registers`, the current state of LLVM IR registers. The size of `Registers` is determined by the function requiring the largest number of LLVM IR registers. Function calls manipulate these registers and the list of stack frames described by `Previous frame`.

A `Field` is a `StateID` to a sub-state, as described in Sect. 2.1. The separation into a root-state and sub-states allows sub-states to grow and the state storage component of DMC, DTREE, to compress them using tree compression [14]. It also allows the use of the delta interface: a write to memory can be simply translated

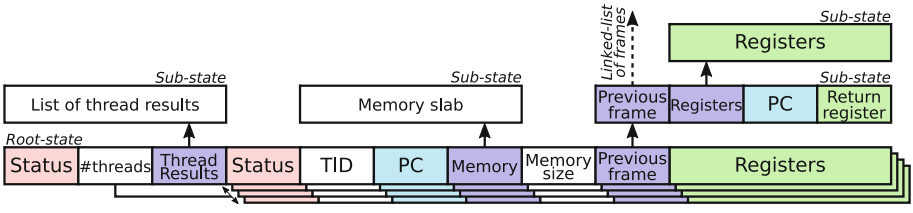

Fig. 3. A description of the state used by LLMC.

to a single, efficient call, taking the current `Memory` index, the offset to write to and the new data. The resulting index can be written to `Memory`.

A single LLVM IR instruction in the program is translated to many LLVM IR instructions in the model. We will distinguish LLVM IR registers in the model from registers in the source program by calling the former *model-registers*. In general, a single LLVM IR instruction is translated to a single step with three phases: In the *Preamble* phase, operands to the source LLVM IR instruction are remapped to model-registers and loaded from `Registers` or `Memory`. In the *Action* phase, the source LLVM IR instruction is cloned, with the operands remapped to the LLVM IR model-registers set up during the Preamble phase. In the *Epilogue* phase, if the source LLVM IR instruction assigns a value to a register, the value returned by the cloned instruction is written to `Registers`.

Listing 1 illustrates how a step is performed as part of the next-state function. Multiple steps can be performed as part of the same transition (line 8), as long as the changes are local to the thread (line 4). This is explained in more detail in Sect. 2.5. The `step` function is called for every thread in the state vector.

2.4.1 Register Manipulation

Note that the `Registers` are not separated into a sub-state, like `Memory`. We chose this such that simple register manipulating LLVM IR instructions would have no need for an indirection and directly translate to an identical instruction, with its operands mapped such that they are loaded from the `Registers` and the return value of the instruction written back to the corresponding register. This allows us to trivially collapse such instructions, combining the Preamble phases, requiring dependencies only to be loaded once.

2.4.2 Memory Instructions

Memory instructions such as loads and stores can be directly mapped to the delta interface, reading or writing only a part of the `Memory` sub-state. There is no distinction between memory allocated on the stack (`alloca`) and on the heap

Listing 1 In the next-state function, the `step` function is called for each thread.

```
1  void step(StateVector sv, int threadID)
2    bool onlyLocal = true; # true while handling commutative instructions
3    bool emit = false;     # set to true when new state is to be emitted
4    while(sv.threads[threadID].pc > 0 and onlyLocal)
5      switch(sv.threads[threadID].pc)
6        case 0: break; # not running, do nothing
7        case SomePC: # PC of first instruction of group
8          # statically collapsed instructions: preamble, action, epilogue
9          # sv.threads[threadID].pc, onlyLocal and emit may change
10       ...
11   if(emit) MC.insert(sv); # emit new state if needed
```

(`malloc`): both allocate memory by growing the `Memory` sub-state. The returned pointer describes which thread created the memory and the offset within the sub-state. Any thread can write to and read from any such memory location. At present, memory cannot be freed, so `free` has no effect. Because of the tree compression, this has no detrimental effect on memory usage, but does mean LLMC currently does not detect `free`-related bugs.

2.4.3 Branching, Function Calls and Threading

To support control flow in LLMC, the `PC` can be changed to the index assigned to the first instruction in the target basic block. If the target basic block contains ϕ-nodes, those registers are updated to the value corresponding to the basic block we are branching from.

Function calls set up a new stack frame with the current `Registers`, `PC` and where to write the return value, then pushes it to the linked list of frames pointed to by `Previous frame`. A return from a function pops the top frame from the list of frames, copies the `Registers` into the state vector, updates the `PC` and writes the return value into the right register. There is no bound on the number of frames; the last frame has `Previous frame` set to 0, indicating no next frame.

Threads are created (`pthread_create`) by enlarging the root state with enough space to fit another thread and incrementing `#threads`. When a thread is done, it is marked as such, but not removed from the state vector. This is to retain the memory allocated by a thread. Due to the compression of DTREE, it has little impact on the memory foot print of the state space. The return value from the thread is added to `Thread results`, where it can be read (`pthread_join`).

2.5 State Space Reduction

Instructions that only have an effect local to a thread do not change the behaviour of another thread. Such instructions are commutative; their respective ordering is not relevant. Thus, such instructions can be collapsed with the previous or next instruction. For example, instructions that read and write only to registers of a thread are local instructions and do not influence another thread. Branching and function calls are other such commutative instructions.

LLMC collapses commutative instructions statically as well as dynamically. The latter is needed to collapse instructions after conditional control flow, because statically the condition is unknown. On-the-fly, the condition is evaluated, the branch taken and it is determined if the next instruction can be collapsed.

2.5.1 Thread-Private Memory LLMC collapses all such commutative instructions, with the important exception of memory operations on memory only accessible to the current thread (memory operations to memory accessible to other threads are never collapsed). This requires knowledge on what memory each thread can access, which LLMC currently does not track. DIVINE implements [4] this by traversing the memory graph in every state, using a run-time

type system to identify pointers and how to follow them (edges); each allocation yields a node.

Nidhugg uses a partial-order-reduction [6] that takes into account from which write a value read by a read originates. In this process, memory operations to thread-private memory are indeed collapsed, because a read can read only a single value: the last value written by the thread itself. The current version of LLMC does not feature an on-the-fly state space reduction for memory operations. Instead, we preprocess the input LLVM IR and statically annotate memory operations that cannot be proven to be local to a thread. While this does reduce the state space, because many operations are to stack variables that remain thread-private, it can only approach the on-the-fly reductions of DIVINE and Nidhugg.

3 Evaluation

Table 1 shows a feature comparison between the tools mentioned in Sect. 1.1. The table shows that RCMC and CBMC do not support dynamic memory in the presence of multiple threads. This limits their usability for our use case, model checking multi-threaded tests of data structures, since numerous thread-safe data structures use dynamic memory. Furthermore, RCMC, CBMC and LLBMC do not support infinite loops and only have limited support for spin-locks. More complex infinite loops like appending a new node in the Michael-Scott queue [17] using `compare-and-swap` are not supported. Thus, we focus on an experimental comparison between LLMC, DIVINE and Nidhugg on execution time, memory footprint of the state space and scalability across multiple threads, since all three tools support using multiple threads for model checking.

Table 1. A feature comparison between the tools mentioned in Sect. 1.1.

			Supported Features									Checks					
Tool	Version	Source	Memory models	Threads	Heap memory	Infinite loops	Global constructors	I/O	Filesystem emulation	`_atomic_*` intrinsics	`atomic {load,store}`	Memory access	`assert()`	Out of Memory	Invalid free	Double free	Memory leaks
DIVINE [3]	4.4.2 (5494190)	LLVM IR	ST[a]	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓
Nidhugg [5]	0.2 (45664bc)	LLVM IR	STPWA[a]	✓	✓	✓	∼[d]				✓	✓	✓	✓	✓	✓	
RCMC [7]	n/a	LLVM IR	(W)RC11	✓	∼[b]	∼[c]	∼[d]			✓	✓					✓	
CBMC [8]	5.10 (ef00f47)	C/C++	STP[a]	✓	∼[b]	∼[c]	∼[d]						✓		✓	✓	✓
LLBMC [9]	2013.1	LLVM IR	n/a		✓	∼[c]	∼[d]						✓		✓	✓	
LLMC [13]	0.1	LLVM IR	STP[a]	✓	✓	✓	✓			✓	✓	✓	✓				
LLMC	0.2 (a732c63)	LLVM IR	S[a]	✓	✓	✓	✓			✓	✓	✓	✓				

[a] Models [16]: **S**) Sequentially consistent; **T**) TSO; **P**) PSO; **W**) POWER; **A**) ARM.
[b] Not supported in combination with threads.
[c] Only trivial spin-locks are supported.
[d] Threads within global constructors not supported.

We ran our experiments on a Dell R930 with 4 E7-8890-v4 CPUs totaling 96 cores and 2 TiB RAM. All sources were compiled using GCC 9.3.0.

3.1 Test Suite

We tested the tools using four real-world concurrent LLVM IR data structures, one concurrent algorithm and one protocol. Sources for all tests are available online[1]. We instantiate the tests with various combinations of threads and number of elements inserted, processed or dequeued. All combinations are listed later, in Table 2. These six tests cover different classes of problem types, different shapes of state spaces, and serve to illustrate the strengths and weaknesses of the tools:

- **SortedLinkedList** ● illustrates a concurrency problem where a number of elements are inserted by a number of threads, with a single outcome: all paths converge to one state. Elements can be inserted throughout the chain.
- **LinkedList** ■, similar to SortedLinkedList ●, but with various outcomes, because the list is not sorted. It has high contention on the head of the chain.
- **Prefixsum** ⬢ is a concurrent approach to determine all sums up to any index in an array. It highlights the ability of the model checker to determine thread-private memory, because the two-pass prefixsum algorithm actually partitions the problem into separate per-thread problems that require no communication and one single-threaded part.
- **Hashmap** ♦ illustrates a concurrency problem where a key is inserted using `compare-and-swap`, followed by either atomically storing the value or busy-waiting on the value, if the key already exists (`findOrPut` [18]). The latter involves atomically loading the value until a non-empty value is loaded.
- **MSQ** ▲ is the well-known Michael-Scott queue [17]. It is similar to LinkedList ■, with the addition of dequeue operations, which may return nothing when the queue is empty. The dequeuer can be made *blocking* by calling `dequeue` until it successfully dequeues an element; this is done in ▲^{2} and ▲^{4}.
- **Philosophers** ▼ is the Dining Philosophers Problem [19], a commonly used protocol to illustrate issues in concurrent resource management. It involves P philosophers and P forks; each philosopher grabs their left fork, then the right, then puts the right fork back, then the left. This is repeated R times. The crux is that each fork is a shared resource for two philosophers. For our tests suite, this illustrates contention on multiple elements in a single array.

These tests highlight the strengths and weaknesses of each tool using real-world data structures and algorithms. The well-known Michael-Scott queue ▲ for example is used in many software packages. They reflect different *kinds* of state spaces: LinkedList ■ focuses on "wide" state spaces, with many end states; SortedLinkedList ● examples state spaces that go wide, but converge into a single end state; Prefixsum ⬢ highlights the model-checker's ability to detect thread-local memory: model checkers that can detect this have a narrow state space, otherwise a model checker will explore all interleavings.

[1] https://github.com/bergfi/llmc/tree/cav2021/tests/performance.

3.2 Observations and Considerations

For each model, we verified that all expected end states were reachable. For example for 1, we manually verified that all $8!/(4!4!) = 70$ possible outcomes of the linked list were generated.

We witnessed DIVINE returning varying state space sizes across different runs on the same test when using multiple threads, indicating a concurrency problem. It also occasionally crashed, most often when using 192 threads. Even though this indicates the answers DIVINE gives might not be correct, we opted to include the results, assuming they would at least provide an indication of the performance.

Furthermore, we did run RCMC on a number of tests. RCMC often runs out of memory before crashing; likely the result of an infinite loop. For even some small tests, it could not finish within 100x the time other tools needed.

3.3 Experimental Results

Figure 4 shows the results of LLMC compared to DIVINE on state space exploration time (4a) and Nidhugg on wall-clock time (4b) when applied to the models from Table 2. These graphs indicate relative performance: the uppermost (blue) line for example indicates the line where LLMC is 100x faster. Figure 4c compares LLMC (lower data points) and DIVINE (upper data points) on the memory compression of the state spaces they generate. Figure 4d compares LLMC (upper data points) and DIVINE (lower data points) on the throughput of states per second.

3.3.1 LLMC vs DIVINE

Looking at the results in Fig. 4a, we see that LLMC outperforms DIVINE by at least 5x in all test cases except Prefixsum ⬢ and two SortedLinkedList ● tests. LLMC suffers in the Prefixsum ⬢ tests because of the lack of dynamic thread-private memory detection. This results in significantly larger state spaces, up to three orders of magnitude for 4, as seen in Fig. 4c.

Comparing the sorted ● and non-sorted ■ linked list cases, we notice LLMC is able to outperform DIVINE in the non-sorted cases by higher factors than the sorted cases. This difference can be explained by that the two tools generate more similarly sized state spaces for non-sorted ■ cases, but not for sorted ● cases. For example, LLMC generates ∼14.4x more states than DIVINE for 4, but only ∼2.2x more for 4. This highlights LLMC is lacking a reduction technique, which works for DIVINE in the sorted cases, but not as well for the non-sorted cases.

For the two Hashmap ♦ cases that both tools completed, LLMC outperforms DIVINE by 8.4x and 157x. Since the hash map is a single global memory object all threads can access, LLMC does not have the disadvantage of lacking a dynamic thread-private memory reduction. DIVINE crashed for the two other ♦ test cases.

DIVINE is unable to complete two of the four Michael-Scott queue ▲ tests, crashing out, the others are verified 86x and 272x faster by LLMC than by DIVINE.

As the complexity of the Philosopher ▼ test cases increases, LLMC increasingly outperforms DIVINE. The two tools generate similarly sized state state spaces,

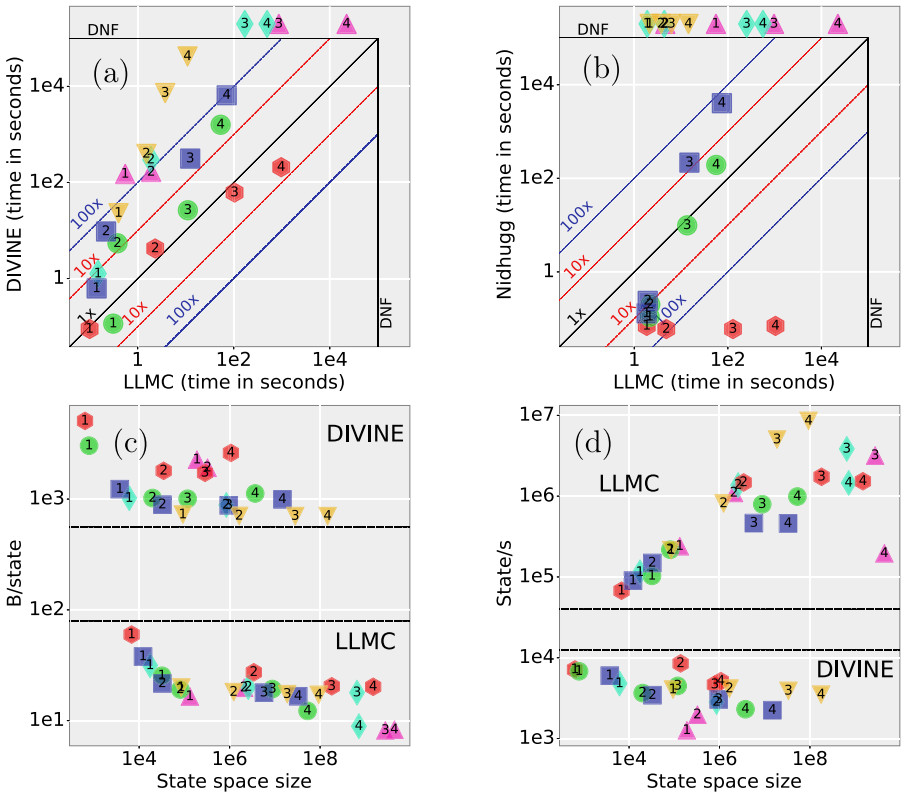

Fig. 4. All experimental results, see Table 2 for a legend. Results above the DNF line mean the tool on the y-axis Did Not Finish, not supporting the test.

because the high contention leaves relatively few memory instructions to be collapsed by DIVINE's reduction, thus levelling the playing field.

In summary, LLMC is able to outperform DIVINE in most of the test cases, mostly between 10x–100x faster, with an outlier as high as 2450x faster (④). This highlights the performance difference, as on average LLMC visits ∼1.4M

Table 2. The six tests with various combinations of number of threads and elements, totaling 24 input programs. MSQ ▲ configurations describe a combination of **E**nqueuers and (**[B]**locking) **D**equeuers in parallel (‖) and sequential (;).

SortedLinkedList			LinkedList			Prefixsum			Hashmap			MSQ	Philosophers	
	Threads	Elements		Thrds	Elems		Thrds	Elems		Thrds	KV-pairs	Configuration	*P*	*R*
①	2	8	①	2	8	①	2	80	①	3	9	▲ E‖E‖D‖D	① 4	2
②	3	6	②	3	6	②	4	80	②	4	12	▲ (E‖E‖E);(D‖D‖D) [B]	② 4	4
③	3	9	③	3	9	③	6	60	③	4	16	▲ E‖E‖E‖D‖D‖D	③ 4	8
④	4	8	④	4	8	④	6	90	④	6	12	▲ E‖E‖E‖D‖D [B]	④ 4	12

states per second (∼8.5M states/s for ▼4), where DIVINE visits ∼4k states per second (Fig. 4d).

3.3.2 LLMC vs Nidhugg

Moving on to Fig. 4b, we notice Nidhugg is unable to complete any of the Michael-Scott queue ▲, Hashmap ♦ or Philosopher ▼ test cases. This is because Nidhugg supports neither the `__atomic_*` instructions needed for the Michael-Scott queue ▲ nor the spin-lock used in the Hashmap ♦ and Philosopher ▼ tests. We tried Nidhugg's transformation capabilities to transform the spin-lock to an assume statement, thus limiting the traces traversed to the ones where the condition of the spin-lock holds, but the generated LLVM IR was invalid and could not be used. Additionally, we tried an experimental version (7b8be8a) with a changelog containing potential fixes to no avail.

We see that Nidhugg outperforms LLMC in the Prefixsum ⬢ test cases consistently by multiple orders of magnitude: Nidhugg traverses only a *single* trace for each of these test cases. This highlights the strength of Nidhugg in its ability to conclude that each read can only read a single value. Without this technique, LLMC needs to exhaustively go through all interleavings of the threads.

For the linked list, sorted ● and non-sorted ■, we see that as the cases get bigger, LLMC is able to outperform Nidhugg. This highlights the disadvantage of stateless model checking: bigger state spaces tend to cause more common prefixes of paths, which causes more work for stateless model checking.

3.3.3 Scalability

Figure 5 shows the results for various number of threads for SortedLinkedList3.9 ●3, chosen for the performance similarity of the three tools. The graph shown is typical: other test expose similar patterns as the one we highlight here. DIVINE does not scale well in the number of threads: its peak performance lies typically around 4 or 8 threads, confirmed by the DIVINE developers[2]. Nidhugg expectedly does scale very well, as threads just execute a specific trace, with hardly and communication. LLMC shows some scalability, but a ∼4x improvement using 192 threads leaves a lot of room for improvement[3].

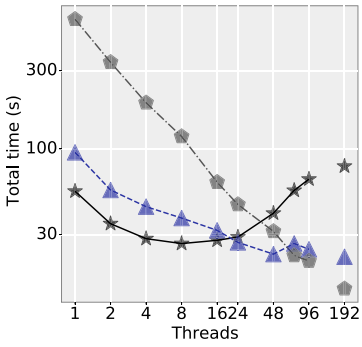

Fig. 5. Scalability comparison of DIVINE ★, LLMC ▲, Nidhugg ⬟.

[2] https://divine.fi.muni.cz/trac/ticket/44.
[3] https://github.com/bergfi/dmc/issues/1.

3.3.4 DMC and Dtree

We highlight one aspect of the performance of LLMC: the underlying model checker DMC and its storage component DTREE [14]. In Figure 4c, we notice that although LLMC on average generates state spaces of an order of magnitude larger compared to DIVINE, it uses two orders of magnitude less memory per state, due to DTREE. Furthermore, DTREE allows to apply a delta to a state without reconstructing the entire state. Since states are typically ∼2kiB in these tests, this significantly avoids copying memory and increases performance.

4 Conclusion

We have introduced LLMC 0.2[4], the multi-threaded low-level model checker that model checks software via LLVM IR. It translates the input LLVM IR into a model LLVM IR that implements the DMC API, the API of the high-performance model checker DMC. This allows LLMC to *execute* the model's next-state function, instead of *interpreting* the input LLVM IR, like DIVINE and Nidhugg. We compared LLMC to these tools using a test suite of 24 tests, covering various data structures. LLMC outperforms DIVINE and Nidhugg up to three orders of magnitude, while other tests have shown areas for improvement. Averaging the results of all completed tests, LLMC is an order of magnitude faster than DIVINE and ∼3.4x faster than Nidhugg. DIVINE and Nidhugg are unable to complete 4 and 12 tests, respectively, due to crashing or not supporting infinite loops or `__atomic_*` library calls.

Future Work. LLMC will benefit most from a state space reduction technique that collapses memory instructions to thread-private memory. We aim to integrate this as part of a memory emulation layer that also adds support for relaxed memory models. Even without the dynamic reduction technique, the results show that LLMC in its current form is a high performing tool to model check software.

References

1. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Lattner, C.: LLVM: an infrastructure for multi-stage optimization. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, IL, December 2002. http://llvm.cs.uiuc.edu
3. Baranová, Z., et al.: Model checking of C and C++ with DIVINE 4. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 201–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_14
4. Rockai, P., Still, V., Cerná, I., Barnat, J.: DiVM: model checking with LLVM and graph memory. J. Syst. Softw. **143**, 1–13 (2018). https://doi.org/10.1016/j.jss.2018.04.026

[4] https://github.com/bergfi/llmc.

5. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_28
6. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proceedings of ACM Program. Lang. **3**(dOOPSLA), 150:1–150:29 (2019). https://doi.org/10.1145/3360576
7. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. **2**(POPL), 17:1–17:32 (2018). https://doi.org/10.1145/3158105
8. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
9. Falke, S., Merz, F., Sinz, C.: The bounded model checker LLBMC. In: Denney, E., Bultan, T., Zeller, A. (eds.) 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, 11–15 November 2013, pp. 706–709. IEEE (2013). https://doi.org/10.1109/ASE.2013.6693138
10. Carter, M., He, S., Whitaker, J., Rakamaric, Z., Emmi, M.: SMACK software verification toolchain. In: Visser, W., Williams, L. (eds.) Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE) Companion. ACM, pp. 589–592 (2016)
11. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
12. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) OSDI, pp. 209–224. USENIX Association (2008)
13. van der Berg, F.I.: Model checking LLVM IR using LTSmin: using relaxed memory model semantics, December 2013. http://essay.utwente.nl/65059/
14. van der Berg, F.I.: Recursive variable-length state compression for multi-core software model checking. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) NFM 2021. LNCS, vol. 12673, pp. 340–357. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76384-8_21
15. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, 10th edn. Wiley (2018). http://os-book.com/OS10/index.html
16. Gharachorloo, K.: Memory consistency models for shared-memory multiprocessors. Stanford, CA, USA, Technical report (1995). https://doi.org/10.5555/891506
17. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Burns, J.E., Moses, Y. (eds) PODC, pp. 267–275. ACM (1996)
18. van der Berg, F.I., van de Pol, J.: Concurrent chaining hash maps for software model checking. In: Barrett, C., Yang, J. (eds.) 2019 Formal Methods in Computer Aided Design (FMCAD), ser. Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 46–54. IEEE, USA, October 2019. https://doi.org/10.23919/FMCAD.2019.8894279
19. Dijkstra, E.W.: Cooperating Sequential Processes, pp. 65–138. Springer, New York (2002). https://doi.org/10.1007/978-1-4757-3472-0_2