Improved State Space Reductions for LTL Model Checking of C & C++ Programs*

Petr Ročkai**, Jiří Barnat and Luboš Brim

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{xrockai,barnat,brim}@fi.muni.cz

Abstract. In this paper, we present substantial improvements in efficiency of explicit-state LTL model checking of C & C++ programs, building on [2], including improvements to state representation and to state space reduction techniques. The improved state representation allows to easily exploit symmetries in heap configurations of the program, especially in programs with interleaved heap allocations. Finally, we present a major improvement through a semi-dynamic proviso for partial-order reduction, based on eager local searches constrained through control-flow loop detection.

1 Introduction

In [2] we have presented an approach to explicit-state LTL model checking of C and C++ programs that make use of POSIX thread APIs for shared-memory parallelism / multi-threading. While the initial implementation already showed promise, it also had multiple shortcomings. We have presented a reduction technique (τ-reduction) that allowed us to successfully model-check small examples, on a scale that would enable model checking of moderately complex unit tests. Nevertheless, the overall performance was unsatisfactory for day-to-day use, due to large state spaces and inefficient interpretation.

The basic approach we follow is to use a C or a C++ compiler with an LLVM-based back-end, such as Clang or GCC/dragonegg to produce, possibly optimised, LLVM bitcode file [17, 18]. Using a modified LLVM interpreter, we then load the bitcode into our parallel LTL model checker DiVinE [3]. We have designed a set of traps that let us create and manage threads, atomicity and a dynamic heap, and on top of these traps, we built a POSIX-compatible thread API.

One of the key advantages of model checking over more traditional verification methods is that it will account for arbitrary thread interleaving, a phenomenon that is very hard to capture in both testing and theorem proving or

* This work has been partially supported by the Czech Science Foundation grant No. GAP202/11/0312.

** Petr Ročkai has been partially supported by Red Hat, Inc. and is a holder of Brno PhD Talent financial aid provided by Brno City Municipality

symbolic model checking. At the same time, unexpected interleavings are a major source of bugs in multi-threaded programs and thus it is extremely desirable to have a tool to help with verification of such programs. Finally, the requirement to deal with an exponential number of interleavings is one of the main problems in implementing a feasible model checker. In addition to systematic exploration of thread interleavings, many model checkers include checking for more elaborate properties than simple safety statements: in case of DiVinE, this entails full LTL_{-X} specification [4].

In our original approach, the proposed τ-reduction allowed us to somewhat restrict thread interleaving without impairing the faithfulness of the model checking process, yielding manageable state spaces for small programs or moderate unit tests. As outlined above, though, this reduction is still not strong enough to facilitate seamless, practical use of a model checker as an integral part of programming effort.

In this paper we identify two major causes of state space inflation in parallel programs and propose more efficient solutions. The first is control flow interleaving, which we discuss in Section 2, the second is memory heap layout, detailed in Section 3. We also take a closer look at the implementation of the interpreter and model checker in Section 5, and finally, we evaluate the new implementation in Section 6, with focus on the reductions described in this paper.

The main contribution of this paper is the combined strength of the newly suggested state space reductions. Using the proposed methods, model checking with DiVinE no longer suffers from the very fine-grained nature of LLVM bitcode. Consequently, regular programming languages, such as C or C++, may be directly used as the modelling language for the model checker, without a prohibitive impact on the size of the state space. The net effect is that the expensive and expert task of manually creating accurate system models can be skipped, turning model checking into a much more accessible method of software verification.

2 Control Flow

The graph induced by a single execution of a deterministic program is a linear sequence of states, with no branching: each state has (at most) one successor. Each "edge" of this induced graph represents a single instruction and each node corresponds to a snapshot of the machine state visible to the program (registers and mapped memory). In a sequential program, this "trace" is identical every time the program is executed with a given input. Without loss of generality, we can assume that input (and any interaction with the environment) is part of the program[1] (an assumption which is actually true in many interesting cases, notably various automated test cases, whether unit, functional or integration).

Generally, a trace that only has single instruction on each edge is more detailed than is useful. A chain of states can be collapsed if they are not relevant

[1] There are other ways to efficiently deal with open-ended inputs and interactivity, most notably symbolic methods. We will discuss these in Section 8 on future work.

for analysis, forming a compound edge which represents an arbitrary instruction block. This technique is known as path compression [24, 15].

However, while any single execution may yield a sequential trace, in parallel programs, the trace may be different every time the program is executed, due to non-determinism inherent in how instructions are scheduled by individual CPUs or cores, and a time-sharing, asynchronous nature of the entire system. This non-determinism is reflected in explicit-state model checkers by introducing branching into the execution trace (which is called a state space in this context), thereby encoding all possible interleavings. In any given state, the system makes a non-deterministic choice on which thread is executed next, creating a single successor state for each active thread. The number of states in the state space is exponential in the number of different threads.

While there are cases where different interleavings produce different end results, there are also many cases where the exact ordering of instructions is irrelevant: different interleavings will yield the same end state. Such confluent executions are redundant and only one of each equivalent set needs to be explored. This idea is at the heart of a class of techniques known as partial-order reductions [20].

In a state space (as opposed to a trace), path reduction can only straightforwardly apply to trace-like sequences of states, where each state has exactly one successor. However, such sub-traces do not naturally occur in state spaces of multi-threaded programs, since almost all states will have multiple successors caused by interleaving. Nevertheless, when a partial order reduction is applied, we choose a single execution among a set of many possible, replacing a diamond-like structure with a trace-like structure. This new trace-like structure is in turn amenable to path reduction, further reducing the number of intermediate states.

Both these reductions can be approximated statically, and one example of such an approximation is the τ-reduction [2]. While its static nature makes τ-reduction extremely simple and easy to implement, it also somewhat limits its effectiveness. In this paper, we introduce a more efficient, semi-dynamic approximation.

2.1 τ+reduction

A simple way to approximate both partial order reduction and path compression is to keep a single thread running as long as cycle and observability criteria are met. In τ-reduction, the observability criterion states that an instruction is observable iff it affects content of shared memory: this approach is inherited without change by τ+reduction. The difference lies in the cycle check. In τ-reduction, any branching (jumping) instruction is treated as possibly closing a control flow cycle, forcing an intermediate state to be generated. However, if we defer the cycle check, we can do much better. Especially in optimised code, branching easily dominates memory access, and the static proviso becomes a major source of inefficiency. In lieu of a simple static check for a branching instruction, we can dynamically detect control-flow loops at successor generation time.

The control location of a thread is kept using a "program counter", a 4-byte integer value that uniquely identifies a specific instruction. Clearly, any actual loop in the program will traverse a single control location twice – hence, it will also encounter the same program counter value. With this in mind, we keep a set of program counter values that we traversed while looking for a successor. Only when an actual control flow loop closes, we interrupt the execution and generate a new state. Each time a successor is generated, the visited set is cleared.

While this is still an approximation, since the (unobservable) loop may finish in finite number of iterations, it is very cheap to compute. Keeping track of full system configurations – an approach that would achieve a better reduction for data-dependent loops with no memory access – would be much more computationally expensive. We reckon that tracking the comparably minuscule program counter value is a viable compromise.

From the model checking perspective, τ+reduction deals with successor states and state spaces, replacing diamonds and chains with one-step transitions. This view is useful for arguing correctness and when thinking in terms of systematic exploration. However, from the point of view of a single execution trace or from the point of view of the program being executed, this view is less appropriate. Therefore, we formulate an alternative, equivalent view of the reduction in terms of interleaving (also called interruption) points.

We define an interleaving point as a place "in-between" two instructions in the program text, where a context-switch (rescheduling) of threads (from the point of view of the program) might happen. When building an unreduced state space, an interleaving point is inserted between each pair of instructions. This intuitively captures what happens in a real CPU, whether a single core time-sharing multiple threads, or an actual multi-core unit. However, as outlined above, not all interleavings cause observable differences in behaviour of the program. τ-reductions then act by removing some of these interleaving points. τ-reduction simply inserts an interleaving point right before each `store` and each branching instruction, statically.

On the other hand, τ+reduction, as a semi-dynamic technique, acts on the program as it is being executed. First, interleaving points are inserted before all `store` instructions, just as with τ-reduction. Then, more are created and removed on the fly: whenever a thread closes a control flow loop, an interleaving point is inserted just before the first instruction that would have been repeated. After the re-scheduling happens, this interleaving point is then dropped again, since a non-looping execution might pass through it at other times. Apart from technical requirement of the model checker that each step is finite, these loop-related interleaving points are intuitively required to avoid delaying other threads indefinitely.

3 Heap

Most non-trivial programs nowadays use dynamic memory, also called a "heap". This memory is allocated on demand using function calls (usually `malloc` and

its variants and `free`) provided by the runtime. The heap allows transparent re-use of memory that is no longer needed, without the requirement to allocate and de-allocate in first in / last out order like with the C stack.

We can consider a heap to be an oriented graph, with nodes representing individual objects and arrows representing pointers. A heap object is a result of a single allocation, it is internally always contiguous, but there is no guarantee on the actual layout of multiple objects in memory. In addition to pointers originating inside heap objects, there may be pointers in stack frames and registers pointing into heap objects (these are known as "root" pointers). While the exact heap layout is irrelevant with regards to program behaviour (bar pointer manipulation or indexing bugs), it affects the actual bit-level representation of a program state.

3.1 Heap Symmetry

This introduces a degree of symmetry into the state space of a program, where multiple distinct states may only differ in heap layout. Since the behaviour of the program is not affected by this difference, we obtain multiple mirror copies of a subset of the state space. This can be extremely wasteful, and is most pronounced when multiple threads are using the heap (which is a common case). Whenever allocations can become interleaved, two symmetric successor states arise, differing only in the ordering of the two heap objects in the physical address space. It is very desirable to detect and exploit this symmetry to reduce the state space.

There are two main ways to implement symmetry reduction. One is based on a modified state comparison function, which detects symmetric situations and makes any two symmetric states equal. The major downside of this approach is that it precludes use of hash tables – the structure of choice in explicit-state model checking. The other option is canonisation: a technique where each state is transformed to obtain a canonic representative of each symmetry class. This way, all symmetric states are represented by the same bit vector, and standard equality and hashing can be used.

On the flip side, detecting symmetric heap configurations is much easier than constructing a canonic representative. This is especially true for programs with explicit (manual) memory management. In some programming languages[2], the heap is subject to automatic garbage collection, and while LLVM has optional garbage collection support, it is not used when compiling C or C++ programs. If exact collection is used [16], all pointers must be tracked by the runtime, especially if using a copying (or more generally, moving) collector. If this information is available, it can be used to implement heap canonisation. In fact, a slightly modified single-generation copying garbage collector will produce a canonic heap layout after every collection cycle.

[2] Or, more exactly, programs, since garbage collection can be implemented for specific programs even in languages without intrinsic garbage collection support.

Opposite to languages with automatic memory management, languages like C and C++ require memory to be explicitly `free`-d to allow memory re-use and avoid resource leaks. However, this also means that the C runtime puts very little constraint on how pointers can be manipulated, since correct memory management is the responsibility of the program, not the system. Unfortunately, this makes it impossible to retrofit garbage collection (and analogically, heap canonisation) to these languages while retaining full generality. In theory, it is legal for a C program to save pointers to a file and read them back later for further use, or to store them bit-flipped in memory or even `xor`'d together as in a `xor`-linked list. Such obscured pointers are however extremely rare in actual programs, and we can make them illegal. Basically, addition is the only reasonable operation to do on an (integer-casted) pointer value; an error can be raised when attempting any other manipulation. In most circumstances, a non-additive operation on a pointer would indicate a bug in the program.

Finally, in a controlled environment (i.e. when each instruction can be freely instrumented), obscured pointers are the only major obstacle in implementing heap canonisation. Therefore, restricting those, it becomes possible to fully track heap pointers throughout the program, and based on this information, compute a canonic heap representation, adjusting all pointers accordingly. The actual layout we chose is based on DFS pre-order, with root pointers forming the initial search stack, global variables first, then deepest frame of the first thread and traversing stacks upwards first, then threads from the lowest thread-id to the highest.

3.2 Tracking pointers

Hence, the remaining problem to solve is exact pointer tracking. While approximate solutions for C and C++ exist, these so-called *conservative* approaches [16] cannot be used for implementing heap reorganisation. A conservative collector will, in a nutshell, treat any bit-pattern as a pointer as long as it corresponds to a valid memory location. Since in a typical program, the heap size is much smaller than the address space and the heap is usually located near its end, this only introduces a small amount of harmless error for a mark&sweep collector, where in the worst case, some garbage is retained. However, a conservative collector must not alter pointers, since it could accidentally alter an integral value that has no relation to the heap, simply having the same bit pattern as a valid pointer.

This means that for successfully tracking pointers, we must use a tagging scheme, where an integer can never be constructed to resemble a pointer and vice versa. On one hand, shrinking pointers by one or two tag bits is not a problem – the address space of the model checker itself is a limiting factor, not the size of a pointer. On the other, it is not feasible to shrink integral types, as this would wreak havoc with established semantics of integer arithmetic[3]. Hence,

[3] This scheme has been adopted in early garbage-collected runtimes, like that of LISP, where all scalars would reserve tagging bits and integer size would not match the machine word size. However, this approach is not feasible in low-level languages.

we cannot easily prevent an integer from mimicking a bit pattern of a pointer. An alternative is to keep tagging information out of band, in a separate image of the address space. This is possible since we can instrument any and all memory access with updates to this tag space at the interpreter level.

All the tracked pointers are created in heap allocations, and their pointer status is preserved throughout their lifetime. We use a special pointer representation, where the heap object and offset into that object are kept apart and manipulated separately. This prevents pointers from overflowing into a neighbouring heap object (this would be a programming error, and must be detected) and makes pointer arithmetic safe and supported. Since programs may not make any assumptions about the bit content of heap pointers, they cannot be legally hijacked for integer constants. Therefore, we can safely rewrite the tracked pointers, without the risk of accidentally altering integral values, or missing actual valid pointers.

Finally, a simple yet efficient optimisation can further reduce the tracking overhead: since we can require and enforce alignment constraints on pointer values, any pointer value will start at a 4-divisible address, thus only requiring a single tracking bit per 4 bytes of memory.

4 Store Visibility

The availability of exact pointer tracking (coming from the implementation of heap symmetry reduction) offers an opportunity to further improve on τ+reduction. In its general form, τ+reduction operates mainly on the notion of "observability": an instruction's effect is a cause for an interleaving point whenever this effect might have been observed by another thread. The main source of observability is writing to (shared) memory: in the thread-based programming model, all memory is implicitly available to all threads. However, it should be noted that in order for a thread to observe a memory write, it must be in possession of a pointer to that memory location.

Therefore, if a memory location has been allocated from the heap by a thread, but the pointer to this heap object is never provided to another thread, this memory location is essentially private to the allocating thread (this most importantly affects `alloca`-obtained memory, see also Section 5.1, although private heap-allocated structures are common as well). Since the layout of heap objects cannot be effectively predicted by the program being verified, it cannot "construct" pointers to objects out of thin air, and they must be explicitly shared by the allocating thread.

Since writes to such "private" heap objects cannot be observed by other threads, we can mark the corresponding `store` instructions as unobservable for the purposes of τ+reduction, again substantially improving its already very good efficiency.

In order to effectively identify the relevant `store` instructions, we trace the root set excluding the currently executing thread. If the heap object that is being written to is not encountered in this manner, then the write is invisible, since no

other thread can read the corresponding memory location. Since we use tracing, this remains true after any combination of `load`s or pointer manipulation. The only action that would make the `store` observable would be a different `store` *in the same thread*, writing a pointer to the relevant object into a pre-existing, already shared memory location. However, since this must happen in the same thread, the change caused by first in such a sequence of `store`s can never be observed, and the later `store` will properly cause an interruption point to be inserted.

5 Implementation

In addition to the new reductions detailed in previous two sections, we have implemented a completely new LLVM bitcode interpreter since [2]. The interpreter itself is a component very important for both robustness and performance of the model checking solution. The previous version of the interpreter was based on the code provided by LLVM itself, with a number of modifications to hook it into the model checking framework. However, this approach had a few disadvantages. First, the interpreter was never built for performance: registers were implemented as arbitrary-precision integers with large space overhead and register files as red-black trees, with cache performance suffering as a consequence.

An explicit-state model checker based on a virtual machine (like our LLVM interpreter), needs to be able to take snapshots of the machine's entire state in order to be able to explore the configuration graph (the state space). These snapshots should be compact and ideally stored as continuous, hashable blocks of memory. In the original interpreter, the states needed to be unpacked into internal data structures and repacked every time a new snapshot was made. On the other hand, the current version takes a different approach, using the compact state representation directly to execute instructions, avoiding expensive unpack/repack operations. Moreover, since most of the data required by the interpreter is packed close together in memory, its cache performance has improved substantially.

5.1 Machine State Vector

A running program on a contemporary commodity computer normally has access to a number of resources. The most important, apart from the CPU itself, is a bulk of random access memory that is traditionally divided into text (program), data, stack and heap. Most systems today, with only a handful of specialised exceptions, do not allow the text of a running program to be modified. In DiVinE, we treat it as constant. Apart from the program text, part of the data region of memory is constant and never modified by the program. This usually entails message strings and numeric constants used in the program. These two sections (text and constant data) are stored only once for each instance of the interpreter. The remainder is stored as a compact *machine state vector*, with layout illustrated by Figure 1.

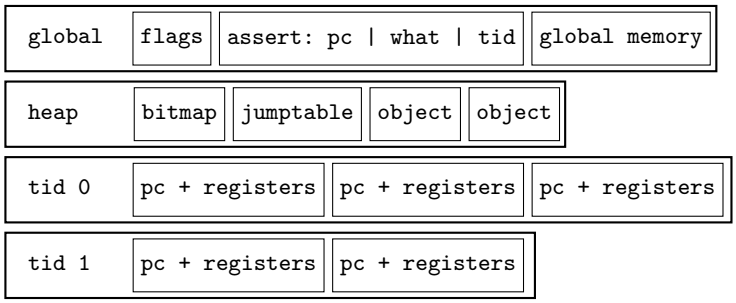

Fig. 1. A state vector with 2 threads, global data and an assertion violation.

Out of the items in a machine state vector, the register stack needs special attention. Real machines (as opposed to virtual) have a limited set of registers, but a (comparatively) unlimited amount of memory. The "stack" in a C program consists of mapped memory and is used for many purposes: saving registers across function calls, storing return addresses and return values, and storing "automatic" local variables. All of this is organised into frames, and each frame on the C stack corresponds to a single entry into a C function.

Contrary to this, the LLVM virtual machine has an unlimited register file. When generating actual executable code, these virtual registers are allocated to machine registers and code for managing register spills (into the C stack) is inserted. However, at the level of LLVM instructions, access to the C stack is provided through the `alloca` instruction and is needed because values stored in registers have no address, and therefore cannot be passed by reference[4].

In our interpreter, we have a structure analogous to C frames, but our frames are not located in memory: they only contain register values and are not addressable (from the point of view of the code being executed). Since LLVM gives no guarantees about layout of memory coming from multiple `alloca` instructions, we allocate `alloca` memory from heap, which in our case is managed automatically. Therefore pointers to `alloca` memory go out of scope when their owner function returns and the heap memory is freed.[5]

5.2 Library Substitution & Masking

Moreover, an important aspect of the software model checking enterprise is API compatibility. In previous versions, we provided a relatively ad-hoc implementation of POSIX threading API, spread over the interpreter itself, but partially

[4] Moreover, until recently, LLVM registers could not hold non-scalar values and those had to be stored in `alloca` or heap memory.

[5] A future revision of the interpreter will release the memory when its owning frame disappears, preventing programs to invoke undefined behaviour. The current version will fail to raise an error in this scenario, since the allocated memory is not specifically bound to its frame.

implemented in a C header file to be included in user code. Our new approach makes a much cleaner separation between "system space" (the interpreter itself and whatever built-in functions – traps – it provides) and "user space" (the user code to be checked and any libraries it links, some possibly provided by DiVinE as replacements for system libraries). Moreover, the separation within user-space is improved as well, since the implementation details of the DiVinE-provided library substitutions no longer leak into the user-supplied code, but are instead linked in at the LLVM level.

Essentially, DiVinE supplies replacements or partial replacements of system libraries, like `libc` and `libpthread`. These replacements are slated to become "drop-in" replacements for their system counterparts, shielding the program under verification from the uncontrolled outside environment. Eventually, such substitution libraries could provide I/O facilities implemented using non-deterministic choice. While this is in principle possible today, a purely-explicit-state representation is ill-suited for verifying programs with significant input-induced branching, especially on large domains.

The user/system-space separation is facilitated by a new technique, implemented through three new traps: `__divine_interrupt_mask`, `__divine_interrupt_unmask` and `__divine_interrupt`. These traps expose a low-level interface to atomicity control, making user-space implementation of library functionality much more feasible. When interrupt masking is in effect, the running thread must not be interrupted by any other thread, until after the masking is lifted. Moreover, the masking is bound to stack frames, which means that there is no danger of leaking the masking into user code, since a `ret` instruction to an originally unmasked function will automatically cause an unmask.

The advantage of explicit atomicity control is twofold: first, it makes library implementation much easier by avoiding the usual pitfalls of writing thread-safe code. Second, it substantially reduces the model checking overhead, since atomicized code is much cheaper to execute, as no intermediate states need to be created. This effect is exponential, since every interleaving point in a library function essentially multiplies the number of states stored during its execution.

5.3 Traps

Apart from the three traps mentioned in previous section, there is a trap, `__divine_choice` which implements non-deterministic choice, and a small set of traps falling into three categories:

1. memory management:
   - `__divine_malloc` – obtain fresh memory from the heap,
   - `__divine_free` – force invalidation of all pointers to an area of memory,
2. thread management:
   - `__divine_new_thread` – create a new thread, with a supplied function as an entry point and a pointer-sized argument
   - `__divine_get_tid` – obtain an identifier of the calling thread,

model	old interpreter		new with heap reduction			
	$\emptyset$	τ	$\emptyset$	τ	$\tau+$	all
peters.c, -O0	OOM	1316162	294193	2181	596	212
peters.c, -O1	148301	11877	33227	491	286	278
peters.c, -O2	89702	6035	21122	443	268	260
pe.bug.c, -O0	OOM	1106757	235272	1617	735	281
pe.bug.c, -O1	221681	19053	49691	613	440	432
pe.bug.c, -O2	188064	14155	43536	613	440	432
fifo.cpp, -O0	fails	fails	559364	22126	1723	108
fifo.cpp, -O1	fails	fails	104642	3926	43	26
fifo.cpp, -O2	fails	fails	83898	2660	148	143
ring.cpp, -O0	fails	fails	2502517	75498	13075	935
ring.cpp, -O1	fails	fails	713743	14157	1461	1405
ring.cpp, -O2	fails	fails	1439424	22735	2121	2065
global.c, -O0	3517	997	451	84	65	26
global.c, -O1	915	179	316	54	30	30
global.c, -O2	887	160	316	54	30	30

Table 1. Number of reachable states in different models under various reductions. The C++ models apparently expose a bug in the old version of the interpreter, hence the numbers are not available. `OOM` means that the model checker ran out of available memory (16GB).

3. and property specification:
   - `__divine_assert` – ensure that a value is non-zero
   - `__divine_ap` – insert an atomic proposition for LTL model checking.

These traps are generally not meant to be used directly in user code, since they have no counterparts in standard system libraries. Instead, they can be used in support code (whether supplied by DiVinE or by the user), which is then linked into the executable before verification. Linking to a different version of the support code (normally, this would entail standard system libraries) will then yield an executable program, directly derived from the verified bitcode file.

5.4 POSIX Threads

We have implemented a substantial subset of the POSIX threading API, including thread management (creation, joining, detaching), mutual exclusion ("fast" and recursive mutexes), condition signalling and thread-local storage. To achieve smooth interoperability, DiVinE provides a `compile --llvm` subcommand, which invokes clang to produce bitcode for verification Since currently DiVinE provides its own `pthread.h` (the interface is not binary-compatible to `glibc` pthreads – some pthread types are shorter in the DiVinE implementation), `divine compile --llvm` will provide the correct `#include` paths and link the program with substitution libraries. This makes producing a verification-ready bitcode file a very easy, single-step operation.

6 Evaluation

To evaluate the actual improvements coming from the proposed reductions, we have taken a small number of example multi-threaded C and C++ programs

	pe.bug.c, -O0	pe.bug.c, -O1	global.c, -O0	global.c, -O1
old, τ	172 -- 193	77	59	n/a
new, τ	54	37	21	19
new, τ+	40	31	18	16
new, τ, store	26	20	12	12
new, τ+, store	12	12	9	9

Table 2. Counterexample lengths for various reductions. The shorter "new, τ" counterexamples are due to better `pthreads` implementation based on masking.

and compared state space sizes (see Table 1) using different options, starting with the old interpreter with no reductions, and with τ-reduction (this was the state of the art at the time of [2]). With the new interpreter, we have made 4 measurements, using heap reduction only, heap and τ-reduction, heap and τ+reduction and finally all reductions including `store` visibility. We have used the following example programs:

- `peterson.c`: a C implementation of Peterson's mutual exclusion,
- `peterson.bug.c`: the same, but with a bug,
- `fifo.cpp`: lock-free first-in, first-out inter-thread queue,
- `ring.cpp`: a lock-free inter-thread ring buffer,
- `global.c`: a race condition when incrementing a shared variable.

While the increased complexity of reductions has non-negligible impact on throughput in terms of states visited per second, this deterioration is much slower than the drop in overall state count. Taking `peterson.c` with `-O1` on a Intel Core 2 Duo P8600 @ 2.4GHz, the throughput ranged from 2650 states/s with no reductions to 780 states/s with all reductions (3.5-fold loss in performance, compared to 530-fold reduction, for 150-fold gain in overall verification speed).

6.1 Counterexamples

A side benefit of the reductions is manifested in counterexample traces. Since the model checker produces a trace consisting of individual states, its length is inversely proportional to length of individual steps. This effect is especially due to the more aggressive τ+reduction[6] and due to introduction of masking in pthread support code. A τ-reduced counterexample trace for the buggy version `peterson.c` spanned several pages, and was very tedious to follow for a human. The τ+reduced version is much shorter and substantially more transparent to users. To put this in a more objective perspective, we have taken 3 sample counterexamples from each of the "buggy" models and summarised the numbers in Table 2.

7 Related Work

Model checking of programs at the level of the source code has been so far pursued in two major research directions. In the first branch, automated code

[6] Symmetric states are very unlikely to appear in a single counterexample, so the contribution of symmetry reduction to this effect is negligible.

extractors were used to replace the error-prone process of manual modelling. Tools such as Feaver [11] or Bandera [8] were introduced to extract C or Java code, respectively, into models to be used as an input for a model checker. The SLAM Toolkit [1] applies the CEGAR approach [6] on top of Boolean programs extracted automatically from a C program source file.

In the other branch, techniques allowing direct analyses of the program source files were examined. Pioneered by VeriSoft [10], model checkers begun to be able to accept C program files as input. While VeriSoft is a state-less model checker, SATABS [7] is a tool performing CEGAR verification of multi-threaded ANSI C programs. Other model checking tools capable of analyzing C programs went into the direction of bounded model checking. While CBMC [5], or F-Soft [14] tools were limited to analysis of sequential programs, TCBMC [21] introduced a bounded model checking approach for POSIX-thread C programs with two threads.

An obvious disadvantage of an interpreter that should be able to read all possible constructs of a high-level programming language such as C or C++, is its structural complexity. Consequently, the model checking tools that directly interpret some high-level programming language, are typically limited to just a subset of it. A possible solution to the problem is to build the tool on top of an intermediate representation language. Recently, the intermediate representation as defined in the LLVM project [17] became rather popular in this respect. Regarding model checking, there are, however, only two tools built on top of the LLVM project. These are, to our best knowledge, the LLBMC tool [19] and DiVinE [2], the former offering SMT-based bounded model checking and the latter enumerative LTL model checking of LLVM bitcode, respectively. Besides LLVM bitcode, Java intermediate representation is used heavily for analysis of Java programs, see Java PathFinder [23].

The key problem of model checking is the state space explosion. When speaking of model checking of the LLVM bitcode, the problem is even more painful as the input language of the model checker is very fine-grained. In this paper we opted to fight this problem using a combination of symmetry [22, 13] and partial order reduction [20].

The problem of unnecessary state space explosion in explicit state model checking due to dynamically allocated entities has been observed and studied before. Symmetry reduction has been applied to model checking of object-based programs that manipulate dynamically created objects. In particular, linear-time heuristic to define canonical representative of a symmetry equivalence class has been presented in [12].

An approach to dynamic partial order reduction somewhat related to our own has been implemented in the VeriSoft state-less model checker [9]. This approach was based on initially exploring an arbitrary interleaving of the various concurrent processes/threads, and dynamically tracking interactions between these to identify backtracking points where alternative paths in the state space need to be explored.

8 Conclusions & Future Work

In our previous work, we have established a baseline which allowed verification of simple C and C++ programs. Most importantly, these programs required no modifications (compared to their form intended for normal execution), in order to be verified.

In this paper, we introduced techniques that significantly move the boundary on feasible verification of such programs. The property-preserving reductions make verification of real-world, multi-threaded C and C++ programs possible and compelling. While we focus on safety verification, in form of assertion statements, invalid memory access or mutex-safety – properties that are easily accessible to everyday programmers without specific verification knowledge – we also provide first-class LTL verification for more expert treatment of mission-critical systems.

There are multiple extensions planned in future revisions. A major class of programs is currently mostly unsuitable for our current approach to verification: open systems with data inputs over non-trivial domains will quickly cause the state space to explode beyond verification capacity of current hardware. Therefore, a semi-symbolic approach for such open-ended programs is required, and at the same time would break a new ground for model checking: programs that employ both multi-threading and data processing are currently out of reach for both explicit-state and symbolic-state tools.

Apart from this, most of the planned future work consists of improvements in the implementation and increasing the coverage of library substitutions to further reduce overhead in verifying new programs. Case-studies from verifying real-world multi-threaded applications are forthcoming as well.

References

1. Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. SLAM2: Static driver verification with under 4% false alarms. In *Formal Methods in Computer-Aided Design (FMCAD'10)*, pages 35–42. IEEE, 2010.
2. J. Barnat, L. Brim, and P. Ročkai. Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs. In *NASA Formal Methods Symposium*, volume 7226 of *LNCS*, pages 252–267. Springer, 2012.
3. J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*, pages 4–7. IEEE, 2010.
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.
5. E. M. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, pages 168–176, 2004.
6. Edmund M. Clarke, Ansgar Fehnker, Zhi Han, Bruce H. Krogh, Joël Ouaknine, Olaf Stursberg, and Michael Theobald. Abstraction and Counterexample-Guided Refinement in Model Checking of Hybrid Systems. *Int. J. Found. Comput. Sci.*, 14(4):583–604, 2003.

7. Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS,05)*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.
8. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *International Conference on on Software Engineering (ICSE'00)*, pages 439–448. ACM, 2000.
9. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, 2005.
10. Patrice Godefroid. Software model checking: The verisoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.
11. Gerard J. Holzmann and Margaret H. Smith. A practical method for verifying event-driven software. In *International Conference on Software Engineering (ICSE'99)*, pages 597–607. ACM, 1999.
12. Radu Iosif. Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 254–261. IEEE Computer Society, 2001.
13. C. Norris Ip and David L. Dill. Efficient Verification of Symmetric Concurrent Systems. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234. IEEE Computer Society, 1993.
14. F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-Soft: Software Verification Platform. In *Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 301–306. Springer, 2005.
15. R. Jhala and R. Majumdar. Path slicing. In *In Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, pages 38–47. ACM Press, 2005.
16. Richard Jones and Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.
17. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*, Palo Alto, California, Mar 2004.
18. http://www.llvm.org/, December 2012.
19. Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In *Verified Software: Theories, Tools, Experiments (VSTTE'12)*, volume 7152 of *LNCS*, pages 146–161. Springer, 2012.
20. D. Peled. All from One, One for All: on Model Checking Using Representatives. In *Computer Aided Verification*, pages 409–423, London, UK, 1993. Springer-Verlag.
21. Ishai Rabinovitz and Orna Grumberg. Bounded Model Checking of Concurrent Programs. In *Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 82–97. Springer, 2005.
22. A. Prasad Sistla and Patrice Godefroid. Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.*, 26(4):702–734, July 2004.
23. Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. Model Checking Programs. In *ASE*, pages 3–12, 2000.
24. K. Yorav and O. Grumberg. Static analysis for state-space reductions preserving temporal logics. *Formal Methods in System Design*, 25(1):67–96, 2004.