Faculty of Informatics, Masaryk University

Memory-Model-Aware Analysis of Parallel Programs

PHD Thesis Proposal

Vladimír Štill

Supervisor:
prof. RNDr. Jiří Barnat, Ph.D.

Brno, 2017

Abstract

Development of parallel programs comes with many pitfalls as it requires reasoning about interaction of threads and safety of communication. Furthermore, testing is not much helpful for bugs dependent on scheduling non-determinism. To make matters worse, contemporary hardware uses relaxed memory semantics: instructions can be reordered by out-of-order execution and memory effects can be further delayed by cache hierarchy. This means that the natural interleaving model of parallelism used both by developers and many formal analysis tools does not expose all possible executions of the program. The problem is further complicated by the fact that different hardware platforms have different memory models which allow various levels of instruction reordering.

This PhD thesis proposal is dedicated to the problem of analysis of parallel programs running on hardware with relaxed memory semantics. It first presents the state-of-the-art in description of memory models and in analysis techniques which take them into account. It further presents goals for the rest of my PhD studies, concretely devising methods for efficient analysis of C and C++ programs running under relaxed memory models. These analysis techniques should be applicable to unit tests of parallel synchronization primitives, data structures, and algorithms. All the techniques will be implemented in the DIVINE model checker. Finally, the thesis proposal summarises my achieved results.

Keywords

relaxed memory, memory models, parallelism, software verification, program transformation, LLVM, C++, DIVINE, implementation

Contents

1 Introduction **1**

2 Relaxed Memory Models **5**
2.1 Description of Memory Model Semantics 6
2.2 Formalized Memory Models 8
2.3 Memory Models and Compilers 15

3 Analysis Techniques for Memory Models **17**
3.1 Decidability and Complexity 17
3.2 Robustness Checking 19
3.3 Direct Analysis Techniques 20

4 Aim of the Work **25**
4.1 Objectives and Expected Results 25
4.2 Time Plan 28

5 Achieved Results **29**
5.1 Published Papers 29

Bibliography **33**

A Publications **43**
A.1 Techniques for Memory-Efficient Model Checking of C and C++ Code 44
A.2 Weak Memory Models as LLVM-to-LLVM Transformations . 59
A.3 Using Off-the-Shelf Exception Support Components in C++ Verification 71
A.4 Model Checking of C and C++ with DIVINE 4 82

Chapter 1

Introduction

Reasoning about the correctness of parallel programs is hard, even if we assume that every memory action a thread performs is visible to all other threads immediately, there is total ordering of these actions, and all loads read from the last write in this ordering. Sadly, this assumption of *sequential consistency* of the memory does not hold in practice as both hardware and compilers perform optimizations which disrupt it. These optimizations include instruction reordering in compilers and out-of-order processors and effects of cache hierarchies in the processors. These techniques are vital for fast execution of all programs, not just parallel ones. In the presence of these relaxations, memory changes can be observed in a different order by different threads. It is the responsibility of the programmer to ensure that the program executes correctly by enforcing ordering of some operations, for example using memory barriers and atomic instructions of a given processor architecture, or using higher level constructs of a given programming language.

In many programming languages, this problem is partially mitigated by the presence of higher level constructs such as mutexes (locks) or synchronized sections of code. These constructs, if used correctly, guarantee that the program will be executed as if running on hardware which preserves sequential consistency. Nevertheless, programmers who design these synchronization constructs, operating systems, and hi-performance parallel data structures have to be aware of memory relaxations arising from the particular memory model.

To complicate matters further, different hardware platforms perform different relaxations of memory accesses – for example, `x86` and `x86-64` (also known as AMD64) processors can only delay stores after loads, while ARM or POWER can also reorder writes with each other and reorder reads with writes arbitrarily (except for reordering of dependent writes). Each platform also comes with a specific set of atomic instructions and memory barriers, which can be used to enforce operation ordering. Therefore, in order to be able to have the same code work on different platforms, it is useful to have support for enforcing memory operation ordering in the programming language itself. This support is also important as the compiler can reorder some operations while it optimizes the code and therefore it must be able to understand constructs that prevent such reordering, so they can prevent it both in the compiler and in the hardware.

Unfortunately, not all programming languages provide primitives related to memory relaxation or even define behaviour of parallel programs. For example, C and C++ had no support for parallel programming until the respective standards from the year 2011. In the older versions, parallelism was achieved only by means of libraries which provided thread manipulation and synchronization primitives (such as `pthreads` on POSIX systems) and memory ordering could have been controlled either by using these synchronization primitives or by compiler-provided language extensions. Apart from the lack of standardized and multi-platform parallel programming support, the problem of this approach is that it is not clear which ordering guarantees arise from the program's code. Other programming languages, such as Java, C#, C11, and C++11, have support for parallelism (including synchronization using mutexes and atomic variables) and their respective specifications describe what guarantees on memory operation ordering these languages provide. It is then the responsibility of the compiler (and virtual machine in the case of Java/C#) to ensure these guarantees are met on any supported platform.

In this situation, we believe that study of memory relaxations all the way from the code in a programming language[1] to the level of the hardware is important for the design of correct data structures and algorithms for parallel programs. Furthermore, we believe this study should produce both descriptions of memory behaviour of programming languages and hardware platforms as well as tools which can help developers who design data structures and algorithms for these platforms.

In my PhD research, I would like to primarily focus on analysis of parallel programs running on hardware with relaxed memory semantics. I would like to explore possibilities of efficient analysis of such programs which would be powerful enough to be usable to developers of hi-performance parallel data structures and algorithms. Such analysis needs to be able to handle unit tests of real-world parallel data structures under relaxed memory models. For these unit tests, it should be able to verify both unreachability of errors, as well as termination and preferably also general liveness properties

[1]By *programming language* we understand higher-level languages in which code is mostly written by humans (e.g. C, C++, and Java) and distinguish them from *assembly languages*, which use platform-specific instructions and syntax, and from *intermediate languages*, which are used in some compilers mainly for platform-independent optimizations (e.g. LLVM IR).

(as given by linear temporal logic). Furthermore, the analysis should be parametrized by the memory model and should support various hardware memory models and the memory model of the programming language. As performance is often critical in parallel programs, I will focus on programs written in C and C++.

Providing a sound and complete decision procedure for memory models is not always possible, as all important problems are undecidable at least for some widespread memory models (more in Section 3.1). Nevertheless, the introduced methods should be designed so that they give high confidence in the correctness of analysed programs. The analysis should primarily be developed for the DIVINE model checker but should also be transferable also to other analysis tools.

The rest of this work is structured as follows: Chapter 2 describes prominent memory models used in both hardware and programming languages. Chapter 3 describes analysis and verification techniques for relaxed memory. These two chapters together give an overview of the state-of-the-art. Chapter 4 then presents aims of my future work and my time plan towards the thesis. Finally, Chapter 5 describes my research results in the area of analysis of parallel programs to date, including results not related to relaxed memory and Appendix A contains selection of my published papers.

Chapter 2

Relaxed Memory Models

The behaviour of a program in the presence of relaxed memory is described by the corresponding relaxed memory model. This memory model depends on the programming language of choice (as it can allow reordering of certain actions for the purpose of optimizations) and on the hardware on which the program is running. It also depends on the compiler (or interpreter or virtual machine) which is responsible for translating the program in a way that it meets the guarantees given in the specification of the programming language. We will abstract from the impact of the compiler and expect it to be correct in most of our considerations. We will also abstract from the impact of an operating system's scheduler which can move program threads between physical processing units, which could be visible in memory behaviour, but the operating system should make sure this effect is not visible.

In hardware, there are two main sources for the relaxed memory behaviour, both of them caused by the fact that memory is several orders of magnitude slower than the processor. One of these sources is the cache hierarchy, which tries to hide speed differences by storing parts of the data in caches. The other is out-of-order execution, which further improves speed by reordering the instructions and issuing instructions speculatively.

Depending on the implementation of these optimizations, different relaxations are observable. On `x86`, only store buffering (delaying of propagation of writes to the memory) is observable, while on ARM or POWER, reordering of all kinds of instructions is observable, as is branch prediction. A more detailed description of causes for memory relaxations (mainly originating from cache hierarchies) can be found in [McK10]. All processors with relaxed memory also provide instructions which allow the programmer to constrain relaxations: memory fences (or barriers) which prevent reordering and atomic instructions such as atomic compare-and-swap or atomic read-modify-write.

When dealing with the memory model of hardware, it is usually neither possible nor useful to discuss the behaviour of a concrete CPU, instead, we discuss the behaviour of a certain platform (e.g. Intel `x86` or IBM POWER). There are at least two good reasons for this: first, results which take into account only the concrete CPU might not be applicable to any other CPU, even from the same family, and second, the exact architecture is usually kept secret by the company manufacturing those CPUs. For this reason, hardware memory models describe processor platforms and should over-approximate the behaviour of processors of a given platform and capture the intent of hardware designers to allow the results to remain relevant even for future processors. The over-approximation might be also needed to simplify the memory model in order to make the subsequent program analysis simpler.

Ideally, formalized memory models of hardware would be produced by the hardware manufacturers themselves, but this is often not the case. Instead, these memory models of contemporary platforms are usually created based on informal descriptions provided by the manufacturers, empirical testing of existing hardware, and discussion with the manufacturers [Sew+10; Sar+11; Flu+16].

Alternatively, one might describe a memory model of a programming language (or a compiler, if the programming language in question does not define memory behaviour of parallel programs). This would then allow the analysis of a program to reason about its behaviour on any platform on which it can be compiled (assuming the compiler is correct). Sadly, similarly to CPU platforms, programming languages usually lack a precise formal description of the memory model, see e.g. [Bat+11] for an analysis of a draft of the C++11 memory model. Furthermore, such specifications can be unnecessarily strict in some cases. For example, according to C++11, any parallel programs in which two threads communicate without the presence of locks or atomic operations have an undefined behaviour and therefore can have arbitrary outcome. Nevertheless, in practice, communication using volatile variables (and possibly compiler-specific memory fences) can work well with most compilers and is often used in legacy code written before C++11 (or C11 in the case of C) where there was no support for concurrency in the language.

In the following sections, we will first look into ways to describe memory models formally (Section 2.1). Then we will inspect important memory models of hardware and programming languages (Section 2.2). Finally, we will shortly discuss the impact of compiler optimizations on memory models (Section 2.3).

2.1 Description of Memory Model Semantics

As already noted, it is often the case that CPU architecture specifications or language specifications describe memory models informally. This can lead to imprecision when such specification is used as a basis for a program, compiler or analyzer implementation. For this reason, it is useful to have formal semantics given to memory models.

The two main options used for the description of memory model semantics are axiomatic semantics, which is usually based on dependency relations between actions of the program, and operational semantics, which describes working of an abstract machine which implements a given memory model.

2.1.1 Axiomatic Semantics

The axiomatic semantics of a memory model usually builds on relations between various memory-related actions of the program and properties of these relations. These relations are mostly partial orders and a sequence of operations usually adheres to a memory model if a union of a memory-model-specific subset of these relations is acyclic (i.e. is a partial order). There are several notations for describing axiomatic semantics, which mostly differ in the names of defined relations and in some details in the description. The framework presented in [Alg+10] aims at description of different memory models in a unified way by a set of common dependency relations. In our figures, we will borrow some notation from this framework, namely the *program order relation* ($\xrightarrow{\text{po}}$) which orders actions performed by a single thread, the *read-from* relation ($\xrightarrow{\text{rf}}$) which connects a read with the store that saved the loaded value, and the *from-read* relation ($\xrightarrow{\text{fr}}$)[1] which connects read with the nearest store after the one read (i.e. with the store which will overwrite the read value). Furthermore, the *write serialization* relation ($\xrightarrow{\text{ws}}$)[2] is notable for describing the guarantee given by all reasonable memory models: for each memory location there is a single total order of all writes to this location. That is, writes to a single location have to be observed in the same order by all threads. Other relations will be introduced as needed in the figures.

2.1.2 Operational Semantics

Alternatively, description of memory models can use operational semantics. Operational semantics describes behaviour of a program in terms of its run on an abstract machine, i.e. by describing the mechanisms which cause memory relaxations (usually in a largely simplified way which should closely match the behaviour of real hardware, but might use very different mechanisms). This usually makes operational semantics easier to understand by programmers and hardware designers and can also lead to a more direct implementation of certain analysis techniques.

[1] In other works also *conflict relation*.

[2] In other works also *coherence relation*.

2.1.3 Other Ways of Describing Memory Models

There are also some works which use different frameworks to describe memory models.

In [AM06] memory models are described in terms of two properties: allowed instruction reordering and *store atomicity*. Store atomicity roughly states that there is a global interleaving of all possibly reordered operations and the authors suggest that it is a desirable property of a memory model. Nevertheless, most architectural memory models lack store atomicity – both SPARC memory models (TSO/PSO/RMO) and `x86`-TSO allow loads to be satisfied from a store buffer, making stores observable in the issuing thread before they can be observed in other threads; POWER further allows independent stores to become visible in a different order in different threads.

The semantics given in [PS16] is based on event structures [NPW79] and considers all runs of the program at once. It is intended to allow reasoning about compiler optimizations. Due to its global view of the program, it is not clear if it can be used for efficient analysis of larger programs.

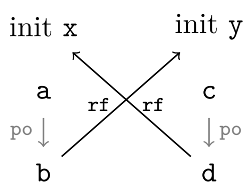
2.2 Formalized Memory Models

In this section we describe commonly used and formalized memory models. These memory models are usually derived from hardware or programming language memory models. In older works, most notable memory models (apart from Sequential Consistency) were memory models of the SPARC processors. These processors can be configured for different memory models (given in order from most strict to most relaxed): Total Store Order (TSO), Partial Store Order (PSO), Relaxed Memory Order (RMO) [SPA94]. Later memory models include Non-Speculative Writes (NSW) memory model presented in [Ati+12]. NSW is more relaxed than PSO but less relaxed than RMO and is notable because the reachability problem of programs with finite state processes under NSW is decidable while for RMO this problem is not decidable. This makes NSW significant even if it does not describe any hardware implementation. Further significant memory models include the `x86` (and `x86-64`) memory model formalized as `x86`-TSO [Sew+10], POWER and ARM memory models, and memory models of certain programming languages, namely Java (Java was the first mainstream programming language with a defined memory model), C#, and C/C++11.

2.2.1 Sequential Consistency

Under sequential consistency (SC) all memory actions are immediately globally visible and therefore can be ordered by a total order (i.e. an execution of a parallel program is an interleaving of actions of its threads) [Lam79]. Furthermore, each load returns the last value written to its memory location in

```
x = 1;   // a          y = 1;   // c
r1 = y;  // b          r2 = x;  // d
```



Reachable `r1 == 0 && r2 == 0`?

Figure 2.1: This code demonstrates behaviour which is allowed under TSO but is not allowed under SC. In the run illustrated on the right, `x = 1` is executed first but the store is buffered and does not reach memory yet. Then `r1 = y` is executed, reading value 0 from `y`. Then the second thread is executed fully and, since the update of `x` was not yet propagated to the memory, it reads 0 from `x`. Finally, the update of `x` (originating from `a`) is performed.

this total order. In the operational semantics, SC corresponds to a machine without any caches and buffers where every store is immediately propagated to the global memory and every load reads directly from the memory. SC is the most intuitive and the strongest memory model and it is often used by program analysers, but it is not widely used in modern hardware. There are no fences in SC, as it has no need for them.

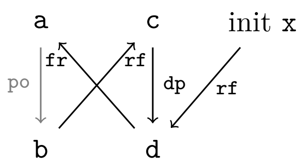
2.2.2 Total Store Order

Total Store Order (TSO) was introduced in the context of SPARC processors [SPA94]. It allows reordering of writes with subsequent reads originating from the same thread. Also, the thread that invokes a read can read value from a program-order-preceding write even if this write is not globally visible yet.

Operational semantics can be described by a machine which has an unbounded, processor-local FIFO store buffer in each processor. Writes are stored into the store buffer in the order in which they are executed. If a read occurs, the processor first consults its local store buffer and if it contains an entry for the loaded address it reads the newest such entry. If there is no such entry in the local store buffer, the value is read from the memory. At any point the oldest value from the store buffer can be removed from the buffer and stored to the memory. This way the writes in the store buffer are visible only to the processor which issued them until they are (non-deterministically) flushed to the memory. Machines which implement TSO-like memory models will usually provide memory barriers which flush the store buffer [McK10; Sew+10].

An example of TSO-allowed run which is not allowed under SC can be found in Figure 2.1.

```
x = 1; // a            while (!g) {} // c
g = 1; // b            r1 = x;       // d
```



Reachable `r1 == 0`?

Figure 2.2: This code demonstrates behaviour prohibited by TSO but allowed by PSO. In this case, the second thread waits for a guard `g` to be set and then attempts to read `x`. However, under PSO, writes to `x` and `g` can be reordered, resulting in action `d` reading from the initial value of `x`. Please note that PSO does not reorder reads and therefore `c` and `d` cannot be executed in inverted order.

2.2.3 `x86`-TSO: `x86` and `x86-64` Processors

The memory model used by `x86` and `x86-64` processors is basically TSO with different fences and atomic instructions than in the SPARC implementation. The memory model is described informally in Intel and AMD specification documents and its formal semantics derived from these documents and experimental evaluation is described in the `x86`-TSO memory model [Sew+10]. The semantics of `x86`-TSO is formalized in HOL4 model and as an abstract machine.

On top of stores and loads which behave as under the TSO memory model, `x86` has fence instructions, a family of read-modify-write instructions, and a compare-exchange instruction.

2.2.4 Partial Store Order

Partial Store Order (PSO) is similar to TSO and was also introduced by the SPARC processors [SPA94]. On top of TSO relaxations it allows reordering of pairs of writes which do not access the same memory location. Operational semantics corresponds to a machine which has a separate store buffer for each memory location. Again, a processor can read from its local store buffers but values saved in these buffers are invisible to other processors [SPA94]. PSO-mode SPARC processors include barriers for restoration of TSO as well as SC [SPA94]. An example of a PSO-allowed run which is not TSO-allowed can be found in Figure 2.2.

This memory model is supported for example by SPARC in PSO mode, but this is not a common architecture and configuration [SPA94; McK10], which means this memory model is mostly important theoretically.

```
x = 1; // a                r1 = x; // c               r3 = y; // e
write_fence();             r2 = y; // d               r4 = x; // f
y = 1; // b
```

Reachable `r1 == 1 && r2 == 0 && r3 == 1 && r4 == 0`?

$$\begin{tikzcd}
\text{init } \mathtt{y} \arrow[dr, "\mathtt{rf}"'] & \mathtt{c} \arrow[d, "\mathtt{po}"'] & \mathtt{a} \arrow[l, "\mathtt{rf}"'] \arrow[d, "\mathtt{ab}"'] & \mathtt{e} \arrow[d, "\mathtt{po}"] & \text{init } \mathtt{x} \arrow[dl, "\mathtt{rf}"] \\
 & \mathtt{d} \arrow[r, "\mathtt{fr}"] & \mathtt{b} \arrow[ur, "\mathtt{rf}"] & \mathtt{f} \arrow[ul, "\mathtt{fr}"] &
\end{tikzcd}$$

Figure 2.3: An example of behaviour allowed by NSW but not allowed by PSO. While the two writes are well ordered, the corresponding reads are not and since the memory model relaxes read-read ordering, they can observe values in different order. The write fence is not necessary, if it was not present the two threads would still not be able to observe different results under PSO, but it is used to demonstrate read reordering more clearly. The fence gives rise to the $\xrightarrow{\text{ab}}$ relation.

2.2.5 Non-Speculative Writes

The Non-Speculative Writes (NSW) memory model, introduced in [Ati+12], is a memory model which is more relaxed than PSO but whose reachability problem for programs with a finite number of finite-state threads is still decidable. The operation model for NSW is also defined in [Ati+12]. It uses two levels of store buffers and a memory history buffer for reordering of reads.

On top of PSO relaxations, NSW allows reordering of reads with other reads and it is defined with read-read and write-write fences and atomic read-modify-write instructions. We show example of NSW behaviour which is not allowed by PSO in Figure 2.3. There are probably no processors which use NSW memory model – it is important theoretically for its decidability proofs.

2.2.6 Relaxed Memory Order

The relaxed memory order (RMO) further relaxes NSW by allowing all pairs of memory operations to be reordered provided they don't access the same memory location [SPA94]. Operational semantics for RMO usually allows instruction reordering in the machine, or involves guessing loaded value at the point of the load instruction and validating the guess later.

RMO is supported by SPARC processors. Examples of other hardware architectures with RMO-like memory models are POWER, ARM, and Alpha [McK10].

2.2.7 POWER Memory Model

POWER is a very weak, RMO-like memory model in which it is possible to observe out-of-order execution as well as various effects of multi-level caches and cache coherence protocols [Sar+11; Mad+12]. For example, POWER allows independent writes to be propagated to different threads in different orders, or loads to be executed before control flow dependent loads (i.e. a load after a branch can be executed before the load which determines if the branch will be taken; this is not possible for writes). An example of POWER-allowed behaviour can be found in Figure 2.4.

The semantics of POWER processors is specified, apart from vendor documents, in both operational and axiomatic formalizations. In [Sar+11] POWER 7 memory model is described in the form of an abstract machine: it is an operational semantics, nevertheless, it is rather complicated due to subtleties of the architecture. This description was later extended in [Sar+12] to support POWER's load-reserve/store-conditional instructions which are used to implement low-level primitives such as compare-and-swap and atomic read-modify-write. An axiomatic semantics of POWER 7 is given in [Mad+12] and also in [Alg+10]. Nevertheless, [Sar+11] observes that while being in agreement with experimental results, [Alg+10] is not matching architectonic intent as well as their operational semantics. To our best knowledge, there is no formal description of the newer POWER 8 or POWER 9 architectures.

2.2.8 ARM Memory Model

The ARM memory model is similar to the POWER memory model, also exposing effects of out-of-order execution and cache hierarchy [Flu+16]. Nevertheless, there are important distinctions between ARM and POWER, both from the point of observable relaxations as well as hardware causes for these relaxations. It was formalized operationally in [Flu+16], building upon the same principles as the operational model for POWER introduced in [Sar+11]. This operational model describes the latest ARMv8/AArch64 64bit architecture and the work compares it to the POWER 7 architecture. There is also an older axiomatic model of ARMv7 given in [AMT14].

2.2.9 Memory Models of Programming Languages

Modern programming languages often acknowledge importance of parallelism and define memory behaviour of concurrent programs. Some programming languages give guarantees that programs which correctly use locks for synchronization observe sequentially consistent behaviour (the *data race free guarantee*). This holds for example for Java [AŠ07] and for the fragment of C++ without atomics weaker than sequentially consistent [TVD14] [ISO12, §1.10.21]. On top of that, some programming languages, such as C, C++,

```
x = 1; // a      y = 1; // b      r1 = x; // c      r3 = y; // e
                                  read_fence();     read_fence();
                                  r2 = y; // d      r4 = x; // f
```

Reachable `r1 == 1 && r2 == 0 && r3 == 1 && r4 == 0`?

$$\begin{tikzcd}
 & & \texttt{a} \arrow[dl, "\texttt{rf}"'] & \texttt{b} \arrow[dr, "\texttt{rf}"] & & \\
\text{init } \texttt{y} \arrow[dr, "\texttt{rf}"'] & \texttt{c} \arrow[d, "\texttt{ab}"'] & & & \texttt{e} \arrow[d, "\texttt{ab}"'] & \text{init } \texttt{x} \arrow[dl, "\texttt{rf}"] \\
 & \texttt{d} \arrow[uurr, "\texttt{fr}"] & & & \texttt{f} \arrow[uull, "\texttt{fr}"'] &
\end{tikzcd}$$

Figure 2.4: An example of behaviour allowed by POWER, but not by NSW. There are 4 threads: two writers writing `x` or `y` and two readers reading both of these variables. The readers observe the updates in an inverted order (i.e. the third thread first reads the new value of `x` and then the old value of `y`, therefore it observes `x` first, but the last thread observes the new value of `y` and then the old value of `x`). The read fences do not help in this case, as the two writes happen in independent threads and therefore are not ordered in any way with respect to each other (the fences are used only to distinguish from NSW). The $\xrightarrow{\text{ab}}$ relation is created by the fences.

and Java provide support for atomic operations which can be used for synchronization without locks if the platform they are running on supports it. C and C++ also support lower-level atomic operations with relaxed semantics which can be faster on platforms with relaxed memory.

C and C++

In C and C++ prior to the 2011 standards, there was no support for threads and shared memory parallelism in the language. In these times creators of parallel programs were dependent on platform and compiler specific libraries and primitives, e.g. the `pthread` library for threading and `__sync_*` family of functions for atomic operations in the GCC and Clang compilers.

The C++11 and C11 standards introduced support for threading and atomic operations to these languages. From the point of relaxed memory models, the interesting part of this is the support for atomic operations and fences.

The atomic operation library provides support for declaration of atomic variables which can be used in atomic operations, such as loads, stores, atomic read-modify-write, and compare-exchange. For any atomic operation, it is possible to specify the required ordering: C/C++ allows not only sequentially consistent atomic operations, but also weaker (low-level) atomic operations which allows implementation of efficient parallel data structures in a platform-independent way.

The C++ memory model is not formalized in C++11 standard, an attempt to formalize it was given in [Bat+11], formalizing the N3092 draft of the standard [ISO10]. While this formalization precedes the final C++11 standard, it seems that there were no changes in the specification of atomic operations after N3092. Nevertheless, there are some differences between the formalization and N3092 (which are justified in the paper).

A notable feature of the C++ memory model is that any program which contains a data race on a non-atomic variable[3] has undefined behaviour. This means that synchronization is possible only by atomic variables and concurrency primitives such as mutexes and condition variables.

LLVM

The LLVM compiler infrastructure [LLV17b] used by the clang compiler comes with its own low-level programming language. The LLVM memory model is derived from the C++11 memory model, with the difference that it lacks release-consume ordering and offers additional *Unordered* ordering which does not guarantee atomicity but makes results of data races defined [LLV17a]. The *Unordered* operations are intended to match semantics of the Java memory model for shared variables [LLV17a].

Java

The Java memory model is rather different from the C++11 one. Its primary goal is to ensure that programs which cannot observe data races under sequential consistency will execute as if running under sequential consistency (the data race free guarantee) [MPA05]. The primary means of synchronization in Java are mutexes (called monitors in Java), synchronized sections of code (which use monitors internally), and volatile variables, which roughly correspond to sequentially consistent atomics in C++11.

Furthermore, as Java strives to be memory safe, it also defines behaviour of programs with data races. This behaviour is rather peculiar, as it is primarily concerned with prohibiting *out-of-thin-air* values – values which, informally speaking, depend cyclically on themselves. These values are primarily prohibited to avoid forging pointers to invalid memory or memory which should be otherwise inaccessible to a given thread [MPA05].

Out-of-Thin-Air Values The problem with out-of-thin-air values is that it is sometimes hard to draw a line between behaviour in which value occurs

[3] Data race is defined as two accesses to the same non-atomic variable, at least one of them being a write, which are not synchronized so that they cannot happen concurrently.

as a result of a well established compiler optimization and where it undesirably occurs out of pure speculation. To that end [MPA05] uses a definition which is based on *justifying executions* – a kind of inductive definition in which more relaxed executions are iteratively built from less relaxed executions. While this semantics intended to allow wide range of optimizations, it later turned out that it disallows certain reasonable optimizations [CKS07; ŠA08; TVD10].

Indeed the task of disallowing out-of-thin-air values while allowing optimizations is hard and there is no consensus on this topic. For example, the C++11 memory model allows these behaviours but at the same time states that implementations are discouraged to exhibit them [Bat+11]. The framework for description of hardware memory models introduced in [Alg+10] disallows out-of-thin-air values based on data and control dependencies. This is too strict for use in programming language memory models as these dependencies are changed by optimizers. It might be acceptable for hardware memory models where dependencies are more explicit and no current hardware exhibits this behaviour, but [Flu+16] mentions that this behaviour is intentionally left allowed by the ARMv8 memory model, in accordance with intents of the hardware architects. An alternative specification of semantics which aims at avoiding this problem was shown in [PS16], proposing new formalization of a fragment of C++11.

2.3 Memory Models and Compilers

When analysing programs in high level programming languages (as opposed to analysing assembly level programs), there can be substantially more relaxation than is allowed by the memory model of the hardware these programs target. The reason is that compilers are allowed to perform optimizations which reorder code or eliminate unnecessary memory accesses. As a result, a compiler can for example merge two loads from a non-atomic variable or assume a load which follows a store to the same memory location to yield the stored value. Further reordering is allowed as per-thread program order is not a total order for programs in languages such as C and C++ (e.g. order of evaluation of function arguments is not fixed by the standard in most cases).

If these optimizations should be taken into account, they complicate the analysis substantially. The two basic options for their handling include reasoning about all permitted reorderings (see e.g. [PS16]), or side stepping the problem by using the same optimizing compiler to produce code both for verification and for actual execution (e.g. by verifying the binary or the optimized intermediate representation of the compiler).

Chapter 3

Analysis Techniques for Memory Models

Analysis and verification of parallel programs is undoubtedly desirable and memory models play an important role in the correctness of these programs; therefore, many techniques exist for analysis of parallel programs under various memory models. In this chapter, we will first look into decidability of common verification problems under relaxed memory models and then we will review some of the approaches to the analysis.

These techniques can be split into two main areas. The first area focuses on verification adherence of a program to a certain memory model if it runs under other, weaker (more relaxed), memory model. For example, they can test whether a given program has no runs under TSO which would not be allowed under SC. Some of these techniques also support fence insertion to restrict behaviours to that of the stronger memory model.

The second category aims at checking correctness of a program (according to some property) under a relaxed memory model (e.g. checking for assertion safety or memory safety or checking of LTL properties). While the first category can be seen as a special case of the second, we consider the distinction notable as the techniques from the first category are usually not used to prove absence of certain types of erroneous behaviour, but assume that all relaxed behaviour is undesirable.

3.1 Decidability and Complexity

Right from the start it is important to note that even if we limit ourselves to programs with a finite number of finite-state threads/processes, there are important problems in widely-used memory models which are not decidable (while reachability under Sequential Consistency is in PSPACE for such programs [SC85]).

3.1.1 Reachability of Error State

Here we ask if the program can reach an error (or goal) state from its initial state. In practice, there can be multiple error states which are given by some property which can be evaluated on each state separately, e.g. we can look for assertion violations or memory errors.

According to [Ati+10], the problem of state reachability in concurrent programs with finite-state processes running under relaxed memory models is decidable for TSO and PSO memory models, but not decidable for RMO (and therefore also not decidable for POWER and ARM). The complexity of the state reachability in these programs under TSO and PSO is non-primitive recursive. In [Ati+12], these decidability findings are further refined: a more relaxed decidable memory model, Non-Speculative Writes (NSW) is identified, and stronger claim about undecidability is proven, showing that adding relaxation that allows reordering reads after subsequent writes to TSO brings undecidability.

The proofs in [Ati+10] use a very simple program model with a finite number of finite-state control units and simple memory actions. Furthermore, they assume that the number of memory locations and processes is fixed and that the data domain is finite. On the other hand, in practice both valid memory locations and processes can be created during the run of the program (and even though there is an upper bound on their number, this upper bound is not practical for the use in analysis).

3.1.2 Verification of Liveness Properties

Liveness properties described by Linear Temporal Logic (LTL) or Computational Tree Logic (CTL) are important class of properties often considered especially in connection with reactive systems and explicit-state model checking [CGP99, Chapter 3]. They allow users to specify properties such as reaction to a certain event or repeated occurrence of an event and they are evaluated on infinite runs of the program. With the automata-based approach to explicit-state model checking these problems are solved by solving repeated reachability of accepting states of Büchi product automaton derived from the program and the specification [BKL08, §5.2].

According to [Ati+10], repeated reachability, which can be used as a basis for verification of LTL properties, is not decidable even for TSO. Furthermore, from the construction of the reduction in the repeated reachability undecidability proof and from [AJ96] it follows that both LTL and CTL model checking problems are undecidable for TSO. Therefore LTL model checking is undecidable for all memory models more relaxed than SC shown in this work. For SC, it is well known that LTL model checking is in PSPACE for finite-state programs [SC85].

3.1.3 Verification of Absence of SC Violations

Here the question is whether a program, when run under a relaxed memory model, exhibits any runs not possible under SC. This problem is explored under many names, e.g. (TSO-)safety [BM08], robustness [BDM13; DM14], and stability [AM11].

Interestingly, [DM14] shows that even for the POWER memory model, checking robustness of programs with a finite number of finite-state threads is in PSPACE, using an algorithm based on reduction to language emptiness. For PSO and TSO, PSPACE algorithm for robustness is shown by [BSS11], this time the algorithm is based on monitoring of SC runs of the program. This shows that checking that a program does not exhibit relaxed behaviour is significantly simpler than checking if this behaviour can actually lead to an error.

3.1.4 Consequences of Decidability and Complexity Results

It can be seen that analysis of a program under a relaxed memory model is a hard task, much harder than for a program running under SC. For this reason most analysis techniques cannot be used to prove the absence of errors, or only in cases when the program is robust to the given memory model. In practice most analysis tools use some kind of constraining of the memory-model-induced reordering: for example bounding the number of instructions which can be reordered, or bounding the number of context switches.

3.2 Robustness Checking

As shown in the previous section, checking robustness (the absence of relaxed behaviour) is significantly less complex than verifying absence of errors in relaxed runs. For this reason, there is an interest in combination of verification under sequential consistency with a robustness checker [BM08]. This way, it is possible to check that program is correct under SC and whether all relaxed runs are equivalent to some SC runs. If both of these checks succeed, it can be concluded that the program is correct under a given relaxed memory model. However, the disadvantage of this technique is that for correctness analysis of parallel programs it can vastly over-approximate possible errors. In practice it is often desirable to allow relaxed behaviours, provided they do not lead to errors: a careful use of relaxed memory can yield much better performance than restricting the program to SC.

In [BM08], the SOBER tool, which allows detection of TSO violations, is presented. This tool works by monitoring sequentially consistent runs of the program and detecting violations which would occur under TSO. The monitoring algorithm is based on vector clocks and axiomatic definition of TSO.

An alternative approach to checking robustness by monitoring SC runs is presented in [BSS11]. This approach allows checking robustness under both TSO and PSO and is built on the operational semantics of these memory models. This monitoring algorithm is implemented in the tool THRILLE and should be asymptotically faster than the one presented in [BM08] while also being sound and complete.

Another possibility for checking TSO robustness is to use *attacks*, a form of restricted out-of-order execution which witnesses SC violations. This approach is presented in [BDM13], together with an implementation in the TRENCHER tool which uses the SC model checker Spin [Hol97] as the backend for validation of attacks.

Restring programs running under the `x86` or POWER memory models to SC behaviours is explored in [AM11]. The work also presents offence, a tool which inserts synchronization into `x86` or POWER assembly to ensure stability.

Concerning stronger memory models, [DM14] shows an algorithm for checking robustness under POWER, but does not provide any implementation. The algorithm presented in this work also assumes that the number of processes is fixed and each process is a finite automaton, therefore it is not directly applicable to robustness checking of real-world programs.

For programming languages with the data race free guarantee,[1] data race freedom can be used as sufficient condition for robustness. The problem of data race detection is explored for example in [YGL04] for the Java Memory Model (JMM). In this case we ask if the program uses enough synchronization to avoid any data races. However, as the JMM defines data races in terms of SC executions, this work only formalizes SC. The entire program, memory constraints, and the specification is encoded as a constraint solving problem, which can be solved by a constraint solver, e.g. Prolog with finite domain data. This work is accompanied by the *DefectFinder* tool.

3.3 Direct Analysis Techniques

Many techniques for safety analysis of programs under relaxed memory models fall into the category of bug finding tools – such tools are unable to prove correctness in general, but they provide substantially better coverage of possible behaviours of parallel program than testing. This incompleteness is mostly caused by either bound on the number of instructions which can be reordered or number of context switches the program can do during any explored run.

[1]Stating that data race free programs observe only sequentially consistent behaviours.

There are several reasons for this bounding; the obvious one is the time complexity of the analysis, but another important reason is that dealing with programs in programming languages is substantially more difficult than dealing with programs represented as a composition of finite-state processes (as assumed in the complexity analyses).

Transformation-Based Techniques

A widely used family of methods for analysis of relaxed memory models is based on transformation of an input program P into a different program P' such that running P' under sequential consistency allows us to explore runs equivalent to running P under a more relaxed memory model. The main advantage of this approach is that it makes it possible to reuse existing analysis tools for sequentially consistent programs together with all the advancements in their development. In most cases the transformation also includes some way of bounding relaxation and therefore this allows exploring only a subset of runs of P. Further under-approximation might be caused by the used SC analyser (e.g. when bounded model checker is used as a backend).

In [Alg+13] a transformation-based technique for the `x86`, POWER, and ARM memory models is presented. This transformation is parametrized and can be tweaked to implement different memory models. It is implemented in the `goto-instrument` tool for instrumentation of `goto`-programs. As `goto`-programs can be created by translation from C, this work primarily focuses on C programs. The output of the transformation is a `goto`-program which can be verified directly by some analysers, or translated back to C. The technique presented in this work is sound, but not complete (due to buffer bounding and possible incompleteness in the backend). It is also not clear if it can cover all cases of delaying reads after writes. The work is accompanied by Coq proofs matching the axiomatic semantics to the operational semantics used for the implementation.

Another approach to program transformation is taken in [ABP11], in this case the transformation uses context switch bounding but not buffer bounding and it uses additional copies for shared variables for TSO simulation. Two options for the transformation are presented, in the first one the total number of context switches is limited, in the second there is a limited number of context switches the value can be delayed for, but the overall analysis is not context-switch-bounded. There is no tool accompanying this publication – the experiments were performed using manually translated C programs.

In [Abd+17] the context-bounded analysis using transformation is applied to the POWER memory model. The resulting program uses nondeterminism heavily to guess the results of a sequence of instructions which is later checked. It uses bounded model checker CBMC as a backend. The publication is accompanied by the `power2sc` tool which implements the transformation of C programs.

Our own work in [ŠRB16b] presents a transformation of LLVM bitcode to simulate buffer-bounded TSO runs. It targets DIVINE and therefore C and C++ programs.

Stateless Model Checking

Stateless Model Checking methods are intended for safety analysis of terminating programs in real-world programming languages [God97]. They employ Dynamic Partial Order Reduction (DPOR) to avoid exploring equivalent runs of the program [FG05] and the works concerning relaxed memory models in this setting often discuss interlay between DPOR and relaxed memory model in length.

A stateless model checking approach to the analysis of programs running under the C++11 memory model (with the exception of release-consume synchronization) is presented in [ND13]. It uses custom implementation of C++ thread and atomic libraries to produce binaries which perform the analysis. It lazily builds relations between memory operations in the form of a *modification order graph*. This representation prevents exploration of infeasible executions as well as unnecessary distinction between equivalent executions. Furthermore, as the C++ memory model allows reordering of reads with future operations, the authors propose to simulate this by propagating stored values to previous loads and validating this speculation (which does not simulate out-of-thin-air values). The paper includes a long discussion on features of the C++ memory model and the corresponding implementation in CDSChecker, which is usable for (small) unit tests of concurrent data structures written in C11 or C++11.

In [ZKW15] the authors focus mostly on modelling of TSO and PSO and its interplay with DPOR. They combine modelling of thread scheduling nondeterminism and memory model nondeterminism using store buffers to a common framework. This is done by adding store buffers to the program and adding shadow thread for each store buffer which is responsible for flushing contents of this buffer to the memory. The proposed approach is implemented in the tool *rInspect*, which is an LLVM-based stateless model checker which supports both unbounded store buffers and buffer bounding (however, as it is a stateless model checker, it works only on programs which terminate).

Another approach to combining TSO and PSO analysis with stateless model checking is presented in [Abd+15]. In this work executions are represented by chronological traces which capture dependencies required to represent interaction between memory actions. These chronological traces are acyclic relations and therefore can be used for DPOR, including the optimal DPOR which explores exactly one execution in the equivalence class of the partial order [Abd+14]. The advantage of this approach is that for robust programs, using the optimal DPOR algorithm with chronological traces should produce the same number of executions under SC as under relaxed memory model. The proposed approach is implemented in an LLVM-based tool Niddhugg, which supports analysis of C programs with pthreads parallelism and with a bounded execution length.

Unbounded Methods

There are also analysis methods which aim to be able to discover any memory-model-related bugs, regardless of number of instructions being reordered or number of context switches.

An approach to verification of programs under TSO, which uses unbouded store buffers, is presented in [LW10]. It uses store buffers represented by automata and leverages cycle iteration acceleration (for cycles involving changes in only one store buffer) to get representation of store buffers on paths which would form cycles if values in store buffers were disregarded. It uses sleep set POR to reduce state space. The provided tool targets a modified Promela language [Hol97]. Since the cycle acceleration is limited to changes in one store buffer, it is not clear if the algorithm is guaranteed to terminate.

Another unbounded approach is presented in [Bou+15] – it introduces TSO behaviours lazily by iterative refinement, and while it is not complete, it should eventually find all errors. This work is based on the robustness checker presented in [BDM13] and uses it to detect runs to which relaxed behaviour should be added. The work is accompanied by an implementation in the tool Trencher.

Other Methods

In [PD95], the SPARC hierarchy of memory models (TSO, PSO, RMO) is modelled using encoding from assembly to Murφ [Dil96]. The encoding allows all reordering of instructions allowed by a given memory model up to a certain reordering bound. This work targets small synchronization primitives such as spin locks.

In [HR06] an explicit state model checker for C# programs (supporting subset of C#/.NET bytecode) which uses the .NET memory model is presented. The verifier first verifies program under SC and then it explores additional runs allowed under the .NET memory model. It can also insert barriers into the program to avoid relaxed runs which violate a given property. The implementation of the exploration algorithm uses a list of delayed instructions to implement instruction reordering. While the authors mention that the number of reordered instructions is not bounded, they do not discuss how this approach works for programs with cycles.

The work [Dan+13] presents an approach for verification of (potentially infinite state space) programs under TSO and PSO using predicate abstraction. The paper first shows that it is not possible to use traditional predicate abstraction to produce a boolean program and then verify this boolean program using weak memory semantics. Instead, they propose a schema which first verifies the program under SC and then extrapolates predicates from the SC run to verify a transformed version of the original program which has store buffers explicitly encoded. The store buffers are bounded in this transformation. Implementation in the tool CUPEX is also provided.

A completely different approach is taken in [TVD14]. This work introduces a separation logic GPS, which allows proving properties about programs using (a fragment of) the C11 memory model. That is, this work is intended for manual proving of properties of parallel programs, not for automatic verification. The memory model is not complete, it lacks relaxed and consume-release accesses. Another fragment of the C11 memory model is targeted by the RSL separation logic introduced in [VN13].

Chapter 4

Aim of the Work

Overall, the aim of my PhD research is to devise methods for efficient analysis of C and C++ programs running under relaxed memory models. These methods should also be implemented and thoroughly evaluated, aiming at real-world usability. Namely, I would like to make it possible to apply relaxed-memory-aware analysis to unit tests of nontrivial parallel data structures and algorithms. The implementation will be primarily working with the DIVINE model checker [Bar+17].

4.1 Objectives and Expected Results

4.1.1 An LLVM-Based Program Transformation for Analysis of Relaxed Memory Models

A large number of verifiers and analysers with support for parallel programs lack support for relaxed memory models and assume sequential consistency. While it is possible to extend these verifiers to relaxed memory models directly in many cases, we believe that an easier and more versatile path lies in transformation of the input formalism for these analysers, as done for example by [Alg+13] or [Abd+17]. This way, the input program is transformed into another program which, when run under SC, simulates runs of the original program under a given relaxed memory model.

The most promising approach seems to be the use of the LLVM Intermediate Representation (LLVM IR) as the source and the target for the transformation. LLVM IR is widely used both by compilers (namely the clang compiler which can be used to compile C, C++, and Objective C on all major operating systems) and by a growing number of analysers with support for parallelism, for example DIVINE [Bar+17], SMACK [RE14], VVT [GLW16], Skink [Cas+17], and Nidhugg [Abd+15]. Furthermore, CPAchecker [BK11] has support for parallelism [BF16] and there are plans to add support for LLVM IR to it. Similarly, CBMC [CKL04] has support for parallelism and planned support for LLVM. Also, LLVM IR can be rather easily transformed as it is used for optimizations in the LLVM framework.

One of the advantages of the program transformation approach is that the same transformation can be used for many analysers. The transformation works by replacing memory operations with either fragments of code or calls to functions which provide implementation of a given operation under a relaxed memory model. This also means that the same transformation, but with different implementations of memory operations, can be used for evaluation of different memory models and modes of their simulation.

An initial LLVM transformation for relaxed memory models was developed for [ŠRB16b] and later extended for [Šti16]. This transformation is now being updated to remove its dependence on a DIVINE-specific API and make its interface more general to work with different memory model implementations.

Furthermore, there are many options in optimization of the transformation, e.g. it is not necessary to transform memory operations for which it can be proven statically that they only access thread-local data. The first of my aims is therefore finishing this program transformation and its optimizations. The transformation will be used as a basis for implementation of memory-model-aware analysis in DIVINE and possibly other verifiers.

4.1.2 An Efficient Support for Non-Speculative Writes Memory Model

The program transformation needs to be accompanied by implementation of memory model operations (memory model runtime). The existing implementations for DIVINE [ŠRB16b; Šti16] support either TSO or a subset of the C++11 memory model without read reordering, both of which use buffer bounding to limit the state space explosion and achieve decidability while keeping the implementation simple.

I would like to implement a framework for simulation of various memory models. The first step in this direction will be to design an efficient operational model for the Non-Speculative Writes memory model. This operational model should be designed so that it can be efficiently implemented and provide good performance for verification.

The NSW memory model was chosen as it is decidable for programs with a finite number of finite-state threads, it is more relaxed than PSO, and it should be possible to implement it reasonably efficiently. To the best of our knowledge, the only operational semantics for NSW is given in [Ati+12] where it is introduced. However, while sufficient for proving its decidability, this semantics is not efficient for verification as it needs to resolve ordering of memory events eagerly, which leads to a lot of branching in the explored state space. It also includes storing complete snapshots of memory in form of history buffers. Instead, we would like to resolve ordering lazily only when actually needed, which should improve scalability of the analysis and to save only relevant parts of memory history.

At first, we will use bounded data structures in the implementation of NSW support. Therefore, the resulting analysis algorithm will not be able to prove the absence of bugs as the number of instructions to be reordered will be bounded (but no other imprecisions will be introduced by this approach). Nevertheless, we believe this approach is reasonable as it can uncover a large number of errors which would otherwise be hard to find.

4.1.3 Heuristically-Directed Exploration Algorithm for Analysis under Relaxed Memory Models

An important aspect of usability of automatic verification and analysis techniques such as model checking is their ability to produce a property violation witness (counterexample) in case a property violation is found. However, usability of these counterexamples depends a lot on the exploration strategy employed by the analyser. For relaxed memory models, it is desirable that counterexamples which contain minimum possible number of deviations from sequential consistency are found first.

Furthermore, it is expected that by directing exploration to find less relaxed runs first, the algorithm will (on average) run faster for programs which contain errors. It might be also possible to employ heuristics to direct relaxations so that relaxed behaviour is first applied on variables on which it is more likely to cause property violations. Another possibility is using robustness-based heuristics and employ relaxed memory semantics only when needed, similar to [Bou+15].

4.1.4 Analysis of Very Weak Memory Models

The POWER and ARM memory models (which are quite similar) are important as they are very weak and there is increasing number of devices which use ARM processors and a good number of hi-performance devices powered by POWER. However, these memory models come with relaxations such as writes which can propagate in different order to different processors and reordering of loads with succeeding writes which can lead to seemingly cyclic dependencies. For this reason, these memory models are more subtle than NSW and require a more advanced analysis.

The C11 and C++11 standards came with a memory model designed to allow for an efficient multi-platform implementation of parallel primitives, even on very relaxed platforms such as POWER/ARM. Therefore, the C++11 memory model is as over-approximation of the POWER/ARM memory models in the context of C/C++ programs, in the sense that all behaviours possible under POWER/ARM are also possible under the C++11 memory model. A very similar memory model is also used by the LLVM intermediate language. As DIVINE is an analyzer for C/C++, it is natural to have support for verification of programs under this memory model.

4.1.5 Techniques for Unbounded Memory Model Analysis

Up to this point I expect to allow only bounded instruction reordering. However, in order to increase coverage of our analysis, I would like to investigate techniques which allow unbounded reordering. Such techniques could use some form of symbolic encoding of delayed memory operations, such as automata-based encoding introduced in [LW10] (which supports only TSO), or they could use abstractions. Another possibility is using SMT-based symbolic encoding. All of these approaches will likely also require changes to the verification algorithm and therefore will not be implemented purely as program transformations accompanied by memory model runtime.

4.2 Time Plan

The plan of the rest of my PhD study and research activities is following:

Now – January 2018 Extension of the relaxed memory support in DIVINE to the NSW memory model and design of verification-friendly semantics for NSW.

January 2018 Doctoral exam and defense of this thesis proposal.

February 2018 – June 2018 Development of heuristically directed search algorithm for verification under relaxed memory models in DIVINE.

June 2018 – November 2018 Extension of relaxed memory support to more relaxed memory models such as C++, POWER and ARM memory models, including development of transformation-friendly semantics of these memory models.

December 2018 – July 2019 Investigation and design of techniques for unbounded verification of programs running under relaxed memory models.

August 2019 – January 2020 Text of the PhD thesis.

January 2020 The final version of the thesis.

Chapter 5

Achieved Results

My work so far has been mostly concerned with analysis of parallel programs and the DIVINE model checker [Bar+17]. It started during my bachelor studies with techniques for compression of state space, which resulted in the publication [RŠB15]. During my master's study, my work included heuristics for state space exploration [ŠRB14] and transformations of LLVM Intermediate Representation [ŠRB16b; Šti16]. These transformations included optimizations which can lead to more efficient verifications and transformations for relaxed memory models.

During my PhD work, I first focused mostly on general verification of parallel C and C++ programs. This included revised support for the C++ exceptions in DIVINE [ŠRB17] and a lot of work on the new version of DIVINE which mostly had character of implementation and resulted in a tool paper [Bar+17].

5.1 Published Papers

- Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. "DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs". In: *Computer Aided Verification*. Vol. 8044. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 863–868. DOI: 10.1007/978-3-642-39799-8_60 [Bar+13]

  Tool paper for DIVINE 3, I have minor contribution to the implementation of DIVINE 3 as described in this paper.

- Vladimír Štill, Petr Ročkai, and Jiří Barnat. "Context-Switch-Directed Verification in DIVINE". in: *Mathematical and Engineering Methods in Computer Science*. Vol. 8934. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 135–146. DOI: 10.1007/978-3-319-14896-0_12 [ŠRB14]

This paper shows that directing search of an explicit-state model checker to first explore runs with low number of context switches can improve performance of the verifier as well as the counterexamples. I have made implementation and evaluation for this paper as well as written part of the text. I have also presented this paper on the MEMICS 2014 conference.

- Petr Ročkai, Vladimír Štill, and Jiří Barnat. "Techniques for Memory-Efficient Model Checking of C and C++ Code". In: *Software Engineering and Formal Methods*. Vol. 9276. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 268–282. DOI: 10.1007/978-3-319-22969-0_19 [RŠB15]

  This paper describes techniques which lead to better memory efficiency of verification of parallel programs in an explicit-state model checker. These techniques include a tree-based compression scheme for state space storage and a custom allocation schema. I have made part of the implementation (concerning the compression), full evaluation and part of the text. I have also presented this paper on the SEFM 2015 conference.

- Jiří Barnat, Petr Ročkai, Vladimír Štill, and Jiří Weiser. "Fast, Dynamically-Sized Concurrent Hash Table". In: *Model Checking Software (SPIN 2015)*. Vol. 9232. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 49–65. DOI: 10.1007/978-3-319-23404-5_5 [Bar+15]

  This paper describes efficient design of a concurrent hash table used in DIVINE. I have minor contributions to this paper.

- Vladimír Štill, Petr Ročkai, and Jiří Barnat. "Weak Memory Models as LLVM-to-LLVM Transformations". In: *Mathematical and Engineering Methods in Computer Science, Revised Selected Papers*. Vol. 9548. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 144–155. DOI: 10.1007/978-3-319-29817-7_13 [ŠRB16b]

  This paper describes the approach to analysis of programs under the TSO memory model using LLVM transformation. I am the main author of this paper, I have made most of the design and implementation, full evaluation, and most of the text. I have also presented this paper on the MEMICS 2015 conference.

- Jiří Barnat, Ivana Černá, Petr Ročkai, Vladimír Štill, and Kristína Zákopčanová. "On Verifying C++ Programs with Probabilities". In:

ACM Symposium on Applied Computing. 2016, pp. 1238–1243. DOI: 10.1145/2851613.2851721 [Bar+16]

This paper describes chaining of DIVINE (which was extended to allow annotation of edges with probabilities) with the PRISM model checker to allow probabilistic analysis. I have provided small part of the implementation (concerning export of state space from DIVINE) and text concerning this part for the paper.

- Vladimír Štill, Petr Ročkai, and Jiří Barnat. "DIVINE: Explicit-State LTL Model Checker". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 920–922. DOI: 10.1007/978-3-662-49674-9_60 [ŠRB16a]

  Competition contribution for SV-COMP 2016 [Bey16]. This paper shortly describes DIVINE and the specifics of applying it to the concurrency category of SV-COMP. I am the main author of this paper, I have written most of the text as well as implemented all modifications of DIVINE which were needed for participation in SV-COMP 2016. I have also had a short presentation of this paper in the SV-COMP session of the ETAPS/TACAS 2016 conference.

- Jan Mrázek, Martin Jonáš, Vladimír Štill, Henrich Lauko, and Jiří Barnat. "Optimizing and Caching SMT Queries in SymDIVINE". in: *Tools and Algorithms for the Construction and Analysis of Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 390–393. DOI: 10.1007/978-3-662-54580-5_29 [Mrá+17]

  Competition contribution for SV-COMP 2017 [Bey17]. This paper shortly describes the SymDIVINE tool which combines explicit and symbolic approach to verification of parallel programs. I have made minor contributions to this paper.

- Vladimír Štill, Petr Ročkai, and Jiří Barnat. "Using Off-the-Shelf Exception Support Components in C++ Verification". In: *Software Quality, Reliability and Security (QRS).* IEEE, July 2017, pp. 54–64. DOI: 10.1109/QRS.2017.15 [ŠRB17]

  This paper describes the approach we took towards verification of C++ code with exceptions in DIVINE 4. We show that carefully selecting which components of existing implementations and libraries to reuse and which to reimplement allowed us to provide full C++ exception support in DIVINE without much cost in terms of runtime performance, implementation effort or increase of complexity of the verifier.

I am the main author of this paper: I have written most of the text and implementation for exception support in DIVINE 4 as well as performed the evaluation for this paper. I have also presented this paper on the QRS 2017 conference. The paper and its presentation was awarded best paper award.

- Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. "Model Checking of C and C++ with DIVINE 4". In: *Automated Technology for Verification and Analysis*. Vol. 10482. Lecture Notes in Computer Science. 2017. DOI: `10.1007/978-3-319-68167-2_14` [Bar+17]

  Tool paper describing architecture of DIVINE 4 and new features of this version. I have written most of the text for this paper.

Bibliography

[Abd+14] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. “Optimal Dynamic Partial Order Reduction”. In: *Principles of Programming Languages*. POPL ’14. San Diego, California, USA: ACM, 2014, pp. 373–384. DOI: 10.1145/2535838.2535845.

[Abd+15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. “Stateless Model Checking for TSO and PSO”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 353–367. DOI: 10.1007/978-3-662-46681-0_28.

[Abd+17] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. “Context-Bounded Analysis for POWER”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 56–74. DOI: 10.1007/978-3-662-54580-5_4.

[ABP11] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. “Getting Rid of Store-Buffers in TSO Analysis”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 99–115. DOI: 10.1007/978-3-642-22110-1_9.

[AJ96] Parosh Aziz Abdulla and Bengt Jonsson. “Undecidable verification problems for programs with unreliable channels”. In: *Information and Computation* 130.1 (1996), pp. 71–90.

[Alg+10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. “Fences in Weak Memory Models”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 258–272. DOI: 10.1007/978-3-642-14295-6_25.

[Alg+13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. “Software Verification for Weak Memory via Program Transformation”. In: *European Symposium on Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 512–532. DOI: 10.1007/978-3-642-37036-6_28.

[AM06] Arvind Arvind and Jan-Willem Maessen. “Memory Model = Instruction Reordering + Store Atomicity”. In: *ACM SIGARCH Computer Architecture News* 34.2 (May 2006), pp. 29–40. ISSN: 0163-5964. DOI: 10.1145/1150019.1136489.

[AM11] Jade Alglave and Luc Maranget. “Stability in Weak Memory Models”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 50–66. DOI: 10.1007/978-3-642-22110-1_6.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. “Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory”. In: *ACM Transactions on Programming Languages and Systems* 36.2 (July 2014), 7:1–7:74. ISSN: 0164-0925. DOI: 10.1145/2627752.

[AŠ07] David Aspinall and Jaroslav Ševčík. “Formalising Java’s Data Race Free Guarantee”. In: *Theorem Proving in Higher Order Logics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 22–37. DOI: 10.1007/978-3-540-74591-4_4.

[Ati+10] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. “On the Verification Problem for Weak Memory Models”. In: *Principles of Programming Languages*. POPL ’10. Madrid, Spain: ACM, 2010, pp. 7–18. DOI: 10.1145/1706299.1706303.

[Ati+12] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. “What’s Decidable about Weak Memory Models?” In: *European Symposium on Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 26–46. DOI: 10.1007/978-3-642-28869-2_2.

[Bar+13] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. “DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs”. In: *Computer Aided Verification*. Vol. 8044. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 863–868. DOI: 10.1007/978-3-642-39799-8_60.

[Bar+15] Jiří Barnat, Petr Ročkai, Vladimír Štill, and Jiří Weiser. “Fast, Dynamically-Sized Concurrent Hash Table”. In: *Model Checking Software (SPIN 2015)*. Vol. 9232. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 49–65. DOI: 10.1007/978-3-319-23404-5_5.

[Bar+16] Jiří Barnat, Ivana Černá, Petr Ročkai, Vladimír Štill, and Kristína Zákopčanová. “On Verifying C++ Programs with Probabilities”. In: *ACM Symposium on Applied Computing*. 2016, pp. 1238–1243. DOI: 10.1145/2851613.2851721.

[Bar+17] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. “Model Checking of C and C++ with DIVINE 4”. In: *Automated Technology for Verification and Analysis*. Vol. 10482. Lecture Notes in Computer Science. 2017. DOI: 10.1007/978-3-319-68167-2_14.

[Bat+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. “Mathematizing C++ Concurrency”. In: *Principles of Programming Languages*. POPL ’11. Austin, Texas, USA: ACM, 2011, pp. 55–66. DOI: 10.1145/1926385.1926394.

[BDM13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. “Checking and Enforcing Robustness against TSO”. In: *European Symposium on Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 533–553. DOI: 10.1007/978-3-642-37036-6_29.

[Bey16] Dirk Beyer. “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*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 887–904. DOI: 10.1007/978-3-662-49674-9_55.

[Bey17] Dirk Beyer. “Software Verification with Validation of Results”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 331–349. DOI: 10.1007/978-3-662-54580-5_20.

[BF16] Dirk Beyer and Karlheinz Friedberger. “A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker”. In: *Mathematical and Engineering Methods in Computer Science*. 2016, pp. 61–71. DOI: 10.4204/EPTCS.233.6.

[BK11] Dirk Beyer and M. Erkan Keremoglu. “CPAchecker: A Tool for Configurable Software Verification”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16.

[BKL08] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of model checking*. MIT press, 2008.

[BM08] Sebastian Burckhardt and Madanlal Musuvathi. “Effective Program Verification for Relaxed Memory Models”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 107–120. DOI: 10.1007/978-3-540-70545-1_12.

[Bou+15] Ahmed Bouajjani, Georgel Calin, Egor Derevenetc, and Roland Meyer. “Lazy TSO Reachability”. In: *Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 267–282. DOI: 10.1007/978-3-662-46675-9_18.

[BSS11] Jabob Burnim, Koushik Sen, and Christos Stergiou. “Sound and Complete Monitoring of Sequential Consistency for Relaxed Memory Models”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 11–25. DOI: 10.1007/978-3-642-19835-9_3.

[Cas+17] Franck Cassez, Anthony M. Sloane, Matthew Roberts, Matthew Pigram, Pongsak Suvanpong, and Pablo Gonzalez de Aledo. “Skink: Static Analysis of Programs in LLVM Intermediate Representation”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 380–384. DOI: 10.1007/978-3-662-54580-5_27.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-03270-8.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15.

[CKS07] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. “The Java Memory Model: Operationally, Denotationally, Axiomatically”. In: *European Symposium on Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 331–346. DOI: 10.1007/978-3-540-71316-6_23.

[Dan+13] Andrei Marian Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. “Predicate Abstraction for Relaxed Memory Models”. In: *International Static Analysis Symposium*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 84–104. DOI: 10.1007/978-3-642-38856-9_7.

[Dil96] David L. Dill. “The Murphi Verification System”. In: *Computer Aided Verification*. CAV ’96. London, UK, UK: Springer-Verlag, 1996, pp. 390–393. URL: http://dl.acm.org/citation.cfm?id=647765.735832.

[DM14] Egor Derevenetc and Roland Meyer. “Robustness against Power is PSpace-complete”. In: *Automata, Languages, and Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 158–170. DOI: 10.1007/978-3-662-43951-7_14.

[FG05] Cormac Flanagan and Patrice Godefroid. “Dynamic Partial-order Reduction for Model Checking Software”. In: *Principles of Programming Languages*. POPL ’05. Long Beach, California, USA: ACM, 2005, pp. 110–121. DOI: 10.1145/1040305.1040315.

[Flu+16] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. “Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA”. In: *Principles of Programming Languages*. POPL ’16. St. Petersburg, FL, USA: ACM, 2016, pp. 608–621. DOI: 10.1145/2837614.2837615.

[GLW16] Henning Günther, Alfons Laarman, and Georg Weissenbacher. “Vienna Verification Tool: IC3 for Parallel Software”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 954–957. DOI: 10.1007/978-3-662-49674-9_69. URL: https://doi.org/10.1007/978-3-662-49674-9_69.

[God97] Patrice Godefroid. “Model Checking for Programming Languages Using VeriSoft”. In: *Principles of Programming Languages*. POPL ’97. Paris, France: ACM, 1997, pp. 174–186. DOI: 10.1145/263699.263717.

[Hol97] Gerard J. Holzmann. “The model checker SPIN”. In: *IEEE Transactions on Software Engineering* 23.5 (May 1997), pp. 279–295. ISSN: 0098-5589. DOI: 10.1109/32.588521.

[HR06] Thuan Quang Huynh and Abhik Roychoudhury. “A Memory Model Sensitive Checker for C#”. In: *FM 2006: Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 476–491. DOI: 10.1007/11813040_32.

[ISO10] ISO C++ Standards Committee. *Programming Languages - C++*. Tech. rep. ISO IEC JTC1/SC22/WG21, 2010. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf.

[ISO12] ISO C++ Standards Committee. *Standard for Programming Language C++. Working Draft N3337.* Tech. rep. ISO IEC JTC1/SC22/WG21, 2012. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf.

[Lam79] Leslie Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Transactions on Computers* 28.9 (Sept. 1979), pp. 690–691. ISSN: 0018-9340. DOI: 10.1109/TC.1979.1675439.

[LLV17a] LLVM Project. *LLVM Language Reference Manual.* 2017. URL: http://llvm.org/docs/LangRef.html (visited on 08/22/2017).

[LLV17b] LLVM project. *The LLVM Compiler Infrastructure Project.* 2017. URL: http://llvm.org/ (visited on 08/22/2017).

[LW10] Alexander Linden and Pierre Wolper. “An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models”. In: *Model Checking Software.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 212–226. DOI: 10.1007/978-3-642-16164-3_16.

[Mad+12] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. “An Axiomatic Memory Model for POWER Multiprocessors”. In: *Computer Aided Verification.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 495–512. DOI: 10.1007/978-3-642-31424-7_36.

[McK10] Paul E McKenney. “Memory barriers: a hardware view for software hackers”. In: *Linux Technology Center, IBM Beaverton* (2010).

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. “The Java Memory Model”. In: *Principles of Programming Languages.* POPL ’05. Long Beach, California, USA: ACM, 2005, pp. 378–391. DOI: 10.1145/1040305.1040336.

[Mrá+17] Jan Mrázek, Martin Jonáš, Vladimír Štill, Henrich Lauko, and Jiří Barnat. “Optimizing and Caching SMT Queries in SymDIVINE”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 390–393. DOI: 10.1007/978-3-662-54580-5_29.

[ND13] Brian Norris and Brian Demsky. “CDSchecker: Checking Concurrent Data Structures Written with C/C++ Atomics”. In: *Object Oriented Programming Systems Languages & Applications.* OOPSLA ’13. Indianapolis, Indiana, USA: ACM, 2013, pp. 131–150. DOI: 10.1145/2509136.2509514.

[NPW79] Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. “Petri nets, event structures and domains”. In: *Semantics of Concurrent Computation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 266–284. DOI: 10.1007/BFb0022474.

[PD95] Seungjoon Park and David L. Dill. “An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order)”. In: *Symposium on Parallel Algorithms and Architectures*. SPAA ’95. Santa Barbara, California, USA: ACM, 1995, pp. 34–41. DOI: 10.1145/215399.215413.

[PS16] Jean Pichon-Pharabod and Peter Sewell. “A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions”. In: *Principles of Programming Languages*. POPL ’16. St. Petersburg, FL, USA: ACM, 2016, pp. 622–633. DOI: 10.1145/2837614.2837616.

[RE14] Zvonimir Rakamarić and Michael Emmi. “SMACK: Decoupling Source Language Details from Verifier Implementations”. In: *Computer Aided Verification*. Cham: Springer International Publishing, 2014, pp. 106–113. DOI: 10.1007/978-3-319-08867-9_7.

[RŠB15] Petr Ročkai, Vladimír Štill, and Jiří Barnat. “Techniques for Memory-Efficient Model Checking of C and C++ Code”. In: *Software Engineering and Formal Methods*. Vol. 9276. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 268–282. DOI: 10.1007/978-3-319-22969-0_19.

[ŠA08] Jaroslav Ševčík and David Aspinall. “On Validity of Program Transformations in the Java Memory Model”. In: *European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 27–51. DOI: 10.1007/978-3-540-70592-5_3.

[Sar+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. “Understanding POWER Multiprocessors”. In: *Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA: ACM, 2011, pp. 175–186. DOI: 10.1145/1993498.1993520.

[Sar+12] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. “Synchronising C/C++ and POWER”. In: *Programming Language Design and Implementation*. PLDI ’12. Beijing, China: ACM, 2012, pp. 311–322. DOI: 10.1145/2254064.2254102.

[SC85] Aravinda P. Sistla and Edmund M. Clarke. “The Complexity of Propositional Linear Temporal Logics”. In: *Journal of the ACM* 32.3 (July 1985), pp. 733–749. ISSN: 0004-5411. DOI: 10.1145/3828.3837.

[Sew+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. “X86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors”. In: *Communications of the ACM* 53.7 (July 2010), pp. 89–97. ISSN: 0001-0782. DOI: 10.1145/1785414.1785443.

[SPA94] CORPORATE SPARC International Inc. *The SPARC architecture manual (version 9)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[ŠRB14] Vladimír Štill, Petr Ročkai, and Jiří Barnat. “Context-Switch-Directed Verification in DIVINE”. In: *Mathematical and Engineering Methods in Computer Science*. Vol. 8934. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 135–146. DOI: 10.1007/978-3-319-14896-0_12.

[ŠRB16a] Vladimír Štill, Petr Ročkai, and Jiří Barnat. “DIVINE: Explicit-State LTL Model Checker”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 920–922. DOI: 10.1007/978-3-662-49674-9_60.

[ŠRB16b] Vladimír Štill, Petr Ročkai, and Jiří Barnat. “Weak Memory Models as LLVM-to-LLVM Transformations”. In: *Mathematical and Engineering Methods in Computer Science, Revised Selected Papers*. Vol. 9548. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 144–155. DOI: 10.1007/978-3-319-29817-7_13.

[ŠRB17] Vladimír Štill, Petr Ročkai, and Jiří Barnat. “Using Off-the-Shelf Exception Support Components in C++ Verification”. In: *Software Quality, Reliability and Security (QRS)*. IEEE, July 2017, pp. 54–64. DOI: 10.1109/QRS.2017.15.

[Šti16] Vladimír Štill. “LLVM Transformations for Model Checking”. Master’s Thesis. Masarykova univerzita, Fakulta informatiky, Brno, 2016. URL: http://is.muni.cz/th/373979/fi_m/.

[TVD10] Emina Torlak, Mandana Vaziri, and Julian Dolby. “MemSAT: Checking Axiomatic Specifications of Memory Models”. In: *Programming Language Design and Implementation*. PLDI ’10. Toronto, Ontario, Canada: ACM, 2010, pp. 341–350. DOI: 10.1145/1806596.1806635.

[TVD14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. "GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation". In: *Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pp. 691–707. DOI: 10.1145/2660193.2660243.

[VN13] Viktor Vafeiadis and Chinmay Narayan. "Relaxed Separation Logic: A Program Logic for C11 Concurrency". In: *Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 867–884. DOI: 10.1145/2509136.2509532.

[YGL04] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. "Memory-Model-Sensitive Data Race Analysis". In: *International Conference on Formal Engineering Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 30–45. DOI: 10.1007/978-3-540-30482-1_11.

[ZKW15] Naling Zhang, Markus Kusano, and Chao Wang. "Dynamic Partial Order Reduction for Relaxed Memory Models". In: *Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: ACM, 2015, pp. 250–259. DOI: 10.1145/2737924.2737956.

Appendix A

Publications

Here I include a selection of my publications in which I have major contribution.

Techniques for Memory-Efficient Model Checking of C and C++ Code[⋆]

Petr Ročkai, Vladimír Štill, and Jiří Barnat

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

Abstract. We present an overview of techniques that, in combination, lead to a memory-efficient implementation of a model checker for LLVM bitcode, suitable for verification of realistic C and C++ programs.
As a central component, we present the design of a tree compression scheme and evaluate the implementation in context of explicit-state safety, LTL and untimed-LTL (for timed automata) model checking. Our design is characterised by dynamic, multi-way adaptive partitioning of state vectors for efficient storage in a tree-compressed hash table, representing the closed set in the model checking algorithm. To complement the tree compression technique, we present a special-purpose memory allocation algorithm with very compact memory layout and negligible performance penalty.

1 Introduction

Model checking is an important verification technique with wide applicability in software development. The older generation of model checking tools primarily targeted special-purpose "modelling" languages, and as such are suitable for stratified, long-term development processes. In those cases, the role of the model checker was towards the early stages, especially in high-level design. However, the trend in the software industry is towards much more tightly integrated development cycles, where all activities are coupled as closely as possible to coding and early deployment. In those scenarios, it would be impractical to add a long and drawn-out process of modelling design elements that are to be programmed (coded) in the implementation language at almost the same time. It is those concerns that motivate the current work on model checking code directly. Additionally, such tight integration of programming and model checking has other benefits: it becomes possible to use the model checker to verify implementation-level properties this way (as contrasted with design-level properties). As such, a sufficiently powerful model checker has the capacity to enter the programmer's

[⋆] This work has been partially supported by the Czech Science Foundation grant No. 15-08772S.

[†] The final publication is available at Springer via `http://dx.doi.org/10.1007/978-3-319-22969-0_19`

toolkit alongside interactive symbolic debuggers (like `gdb`) and runtime analysis tools (like `valgrind`).

While it is quite obvious that those are all worthwhile goals, model checking of executable code presents substantial challenges. In the case of explicit-state model checking, the approach used by the DIVINE model checker [1], those challenges derive from the large number of distinct states reachable through execution of programs. This is most pertinent to multi-threaded programs, where model checking happens to be also most useful. Besides the size of the state space, the primary challenge in verifying a program directly lies in the interpretation of the source code. In DIVINE, this challenge was quite successfully resolved by using a standard C/C++ compiler with an LLVM backend, and by interpreting the resulting bitcode instead of the (much more complicated) original source code. Besides simplifying the implementation of the model checker, this also removes large portion of the complicated C++ compiler from the trusted code base.

The remaining challenges, stemming from large state spaces, are hence twofold: the time required to explore the state space, and the memory required to store it. Some techniques attack both problems at once: reduction techniques that vastly reduce the number of reachable states are one such approach [7]. In this regard, DIVINE employs a very successful τ+reduction [8] which removes many thread interleavings and compresses state chains down to a single transition, without compromising the soundness of model checking. Some approaches target one of those problems specifically: one such is parallelisation, which exclusively aims at reducing the time required for a verification run to complete. This is an important goal because a verification tool that can be used interactively is more valuable than a batch one, where the user needs to wait overnight (or for a week) to obtain the result. In this regard, DIVINE employs parallelism extensively and achieves decent speed-ups through its use.

Finally, despite extensive state space reduction, the state spaces obtained from C (and especially C++) programs are very large, memory-wise. And while parallelism gives us an acceptably fast algorithm, it is easy to run out of available memory. Of course, there is always room for optimisations: the LLVM interpreter embedded in DIVINE is currently the main speed bottleneck, and as such is subject to ongoing optimisation effort. Nonetheless, even in its current incarnation, on most computers, DIVINE will run out of memory very quickly. As such, techniques that reduce memory use are of prime importance, even if they have a modest negative impact on speed.

1.1 Reducing Memory Use

There are a few elements in an explicit-state model checker where large amounts of (fast, random-access) memory are required. Usually, by far the most extensive is the representation of the closed set, although the open set (usually a queue in a parallel model checker) can become quite large as well. The representation of the program being model checked is usually small and of constant size throughout the computation, as is the code of the model checker itself. Hence, for all but very small models, the memory requirements of the model checker are dominated by the open and closed sets, which are composed of state vectors and often some ancillary per-state data of the model checking algorithm. Besides the state vectors themselves, the fact they are organised in a data structure (a hash table, a queue or similar) causes memory overhead of its own. While with "plain" LLVM-based model checking the state vectors are very large (often many kilobytes), and as such, eclipse the memory requirements of all the data structures that hold them, we will see the importance of memory efficiency of those data structures rise in prominence when the amount of memory occupied by a single state vector shrinks considerably.

One important technique that can contribute to memory efficiency of explicit-state model checking is lossless compression. Several methods of lossless compression – including methods based on state vector decomposition – were introduced over the time as discussed in Section 1.2. In our work we present an extension of existing state vector decomposition methods that is particularly well suited for real-world application of model checking of C and C++ code through LLVM bitcode – it supports dynamically sized states, has no need for preallocation of fixed-size closed set and supports parallel model checking. We show in our experiments that for verification of real-world programs with DIVINE, the method we describe constitutes enabling technology. That is, we show that it is possible to verify programs where verification without compression would require terabytes of RAM.

1.2 Related Work

The oldest and simplest lossless compression method was to use a generic data compression algorithm (Huffman coding, arithmetic coding, etc.) to compress individual state vectors before storing them into memory [5, 3]. These approaches only minimally exploit the redundancy *between* different states, which is usually much higher than the redundancy *within* a single state vector.

In this respect, a better method has been proposed in [4], where the state vector is decomposed and each slice of the vector is hashed separately and only indices to those slices are saved as a state. This exploits the fact that many state vectors contain parts that are identical between different states and also much longer than a single pointer – hence, storing a pointer to a separately hashed slice is more memory-efficient than storing the duplicated area repeatedly. While this idea is in a way a specialisation of otherwise very generic and well-known dictionary-based compression (as employed by the commonly used LZ77 [10] algorithm), it has some special properties that make it more interesting for model checking: namely, the construction of the "dictionary" makes it easy and efficient to hash the compressed states and compare them for equality – neither of those steps needs to decompress states already stored.

The one-level scheme proposed in [4] has been improved upon by [2], making it fully recursive. It also removes the requirement that the compression algorithm knows specifics about the state vector layout. This recursive approach has been further adapted for parallel model checking in [6]. One downside of this implementation is a requirement for a fixed-size, pre-allocated hash table with fixed size slots.

We use a similar scheme, but we re-introduce *optional* state vector layout awareness into the compressor, we use generic n-ary trees instead of binary, we use resizing hash tables in the implementation and we focus on dynamically sized states which naturally occur in LLVM-based programs which include memory allocation.

2 Tree Compression

Depending on the verification task, the storage size of a single vertex (state) can be fairly large. This is especially true of more complicated model checking inputs, like timed automata or LLVM[1]. In those cases, it makes sense to consider compression schemes for states and/or the entire state space. In DIVINE, we have implemented the latter [9], using a scheme similar to *collapse* [4]. Since our hash table is resizeable to facilitate better resource use, we cannot directly use some of the improvements that rely on fixed-size hash tables [6]. On the other hand, since the hash table we use can accommodate variable-size keys, we are not limited to fixed-layout trees and can use content-aware state decomposition like in the original *collapse* approach (but unlike original collapse, we can decompose the state recursively, which is useful with more complex state vectors, like those arising from LLVM inputs). The decomposition tree structure is illustrated in Figure 1.

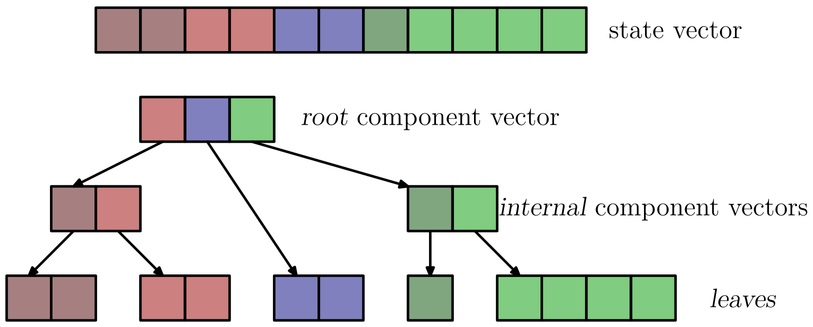


Fig. 1. A decomposition of a state into a component tree. The leaves represent fragments of the original state vector.

Our approach uses three hash tables that are adaptively resized as needed. One holds root elements – one root element corresponds to each visited state

[1] In theory, nothing about LLVM per se causes states to be large; in practice, however, inputs that are expressed in terms of LLVM have a tendency to have much richer state than more traditional formalisms, like DVE or ProMeLa.

1:1. These root elements are represented as component vectors, where each component is represented as a separate object in memory. Those components are de-duplicated using a *leaf table* – a state fragment that is identical in multiple different states is only stored once, and the *root* table refers to the de-duplicated instances of those objects. To facilitate recursive decomposition, we also maintain a third table, *internal*, for internal nodes of the state decomposition tree. The *internal* nodes have the same structure as *root* nodes (a vector of pointers), but they do not correspond to complete states and the *internal* table is not consulted by the model checking algorithm when looking up vertices during search.

The component vectors contain a flag to decide whether a particular component is another component vector or a state fragment, as otherwise they are not distinguishable – both are stored as raw byte arrays in memory, without distinct headers. Clearly, reconstructing a state vector from a component vector is easily done by walking the decomposition tree and copying leaf node content to a buffer from left to right. In theory, storing the size of the entire state in the *root* component vector could improve efficiency by making the reconstruction work in a single pass, copying fragments into a pre-allocated buffer. In practice however, the decomposition trees are small and the requisite pointers are retained in fast CPU cache on the first pass (when the buffer size is computed), making the savings from a single-pass algorithm small. Moreover, the extra memory overhead of storing another integer along with each state is far from negligible.

The trade-off inherent in tree-based compression schemes is visible in Figures 1 and 2. Compare the number of squares (memory cells) in these two pictures. The original state vector occupies 11 cells, its decomposition uses 18 cells. However, adding another similar state (state B in Figure 2) increases the memory use only by 9 cells in the compressed variant, while it would add another 11 cells without compression. The state vectors illustrated here are extremely small; real-world LLVM states typically occupy thousands of memory cells and bigger states naturally favour compression. On the other hand, a realistic implementation introduces slightly more memory overhead than the idealised picture show here.

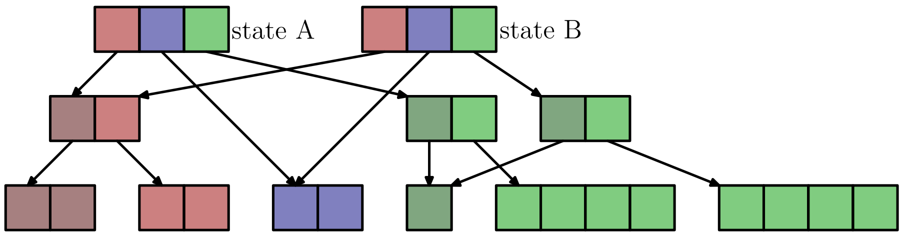


Fig. 2. A de-duplicated pair of states. The layers are analogous to Figure 1. States A and B differ only in the light green component.

2.1 Splitting State Vectors

The fact that both the component vectors in the internal nodes and the state vector fragments stored in the leaf table are of possibly variable size (and making them fixed-size would not improve compactness, thanks to the memory allocator design described in Section 3), we gain the capability to decide on how to split state vectors dynamically. This capability can be used to align boundaries of both leaf fragments as well as their groupings with logical divisions of the state vector. The working hypothesis is that this would improve compression ratio, since changes between state vectors that are neighbours in the state space have a tendency to be localised within the state vector. By correctly aligning the split points for the purposes of compression, we expect the changes between a pair of related state vectors to be localised to the smallest possible subtree. Moreover, the size of a decomposition tree has an impact on performance: if we can identify large contiguous chunks of the state vector that change only rarely, if at all, we can reduce the size of the decomposition trees and thus improve the overall speed of verification. On the other hand, if those larger chunks in fact do change, this has adverse effect on compression ratio. Therefore, finding a good way to split the state vectors is a balancing act: smaller leaf fragments and more balanced trees lead to better compression, but incur higher performance penalty. Of course, leaf fragment size cannot be reduced arbitrarily: to achieve compression, a leaf must be strictly larger than a single pointer (8 bytes), since the reference in the parent node is represented using a pointer.

2.2 Interactions

The tree compression methods interacts with other components of the model checker. First, the memory allocation regime is an important aspect: how big a pointer to a node is, for example, is quite important from the perspective of compression ratio. With 32-bit pointers, compared to 64-bit, we could expect nearly twice the memory efficiency. However, that would also limit the number of nodes in the compression tree to about 4 billion: considering that on realistic x86-class hardware, exploring and storing 40 billion states is possible, and even if we neglect the requirement to also store internal and leaf nodes of the tree, 32-bit pointers are clearly insufficient.

Another aspect to consider is how the requirements of parallel exploration affect the compression method. In shared memory, DIVINE offers two exploration modes, *shared* and *partitioned*. On modern hardware, the *shared* mode is usually faster, especially with higher thread counts. In the context of compression, it offers another important advantage: since it uses a single hash table which is shared by all the workers, tree compression is very efficient. Since all states are stored in the same (compressed) hash table, all redundancy can be exploited for compression. With the *partitioned* scheme, on the other hand, each state is statically assigned to a particular worker thread, and each thread maintains a private hash table. This hash table is slightly more efficient (because access to it does not need to be thread-safe), but this advantage is usually outweighed by more costly communication between the threads which need to exchange states based on the partitioning. The effect on compression is even more pronounced, though: since each thread stores – and compresses – state vectors privately, a large fraction of the leaf and internal nodes will be duplicated. This happens whenever two state vectors share a subtree, but are assigned to different worker nodes. This subtree would only be stored once in the *shared* scheme, but twice in the *partitioned* scheme.

On the other hand, DIVINE also offers a *distributed-memory* mode, using MPI for communication. This mode necessarily works just like the *partitioned* mode in shared memory: each machine in the cluster has a private hash table and compression is performed locally within that hash table. This means the compression will be less efficient in distributed-memory situations, nonetheless substantial savings are still possible.

Finally, besides the closed set stored in the hash table (or hash tables in partitioned and distributed modes), a model checker needs to maintain an open set. In parallel algorithms, both for checking safety (reachability) and for LTL model checking (OWCTY), this is often a queue. Since the compression method we use is lossless, the state vector can be reconstructed from its compressed form and it is possible to also compress the open set, in addition to the closed set.

3 Memory Allocation

Memory allocation is an extremely frequent operation in an explicit-state model checker. Moreover, the memory pool that threads allocate from is a shared resource, requiring certain amount of synchronisation. One way to side-step this issue is to statically pre-allocate as many resources as possible – this is the approach taken by, most prominently, the model checker SPIN. The main downside of this approach is that the tool either has to "guess" resource use very well ahead of time, or rely on the user to provide guidance. In all but very simple scenarios, the former is very hard to get right – models vary wildly from one to another in which parts of the model checker they stress. Some require very long queues or deep stacks, even when the overall size of the state space is comparatively small. Others only need a very small queue but the state space is huge, and almost all memory needs to be allocated towards the closed set. Some models have few big states, requiring few slots in the hash tables, but need a lot of memory for storing the states themselves.[2]

However, there is a more important limitation, namely with regard to multitasking: users expect to be able to execute multiple instances of a program at the same time, especially if the verification runs are well below the limits of the computer they are using. Static resource allocation in such cases becomes a chore – especially so if multiple users are involved on shared hardware. In most cases, we aim at interactive use: batch scheduling is only suitable for very

[2] The LTSmin model checker avoids this particular resource split by storing state vectors decomposed, each fixed-size chunk stored inline in the large pre-allocated hash table.

large instances, where the entire computer (or a cluster) is tied up in a single verification task. Meanwhile, a large SMP system can easily serve many tasks and many users interactively – but this means that tasks should only consume resources that they actually need, so that resource conflicts are minimised. This is very hard to achieve if memory needs to be pre-allocated at a time when the size of the state space is not yet known.

To address those issues, DIVINE uses dynamic allocation for all resources, achieving optimal hardware utilisation when multitasking. There are, however, multiple challenges associated with this flexibility, especially when dealing with parallel algorithms.[3]

3.1 Allocation Profile

When designing a custom memory allocator, the first thing to ask is what is the allocation profile of our target application. Are object sizes similar, or distributed across a wide spectrum? Are there many small allocations, or few big allocations? Is memory retained for a long time, or a short time? Is memory deallocated often?

We can answer most of those questions for DIVINE: for one, there is a tendency to see many objects of similar size. This is most visible in models with fixed-size states (this is actually the case with majority of input languages in DIVINE: most traditional modelling languages require all state variables to be explicitly declared and do not provide dynamic variables). It is also true, to a smaller extent, with variable-size state vectors: many states will differ in content but not the size of the state vector. For LLVM, state size changes when a thread is created, a function is entered or left and when a new thread is created or when heap memory is allocated. All these operations are comparatively rare, so we can expect many states of any given size to appear over time. This is even more pronounced when compression enters the picture, since the fragments have more uniform sizes than the entire state vectors. This favours a design where objects of a particular size are grouped into bigger blocks, reducing overheads in the parent allocator (both time and memory overhead).

This type of layout also offers the opportunity to store exact object size as allocator metadata, once per block of objects. When state vectors (or their fragments) are of variable length, their length needs to be stored somewhere: if each state vector stores its own length, this either adds 4 bytes of overhead per state (or, when using 2 bytes, causes the rest of the vector to be stored unaligned which incurs a large performance penalty). Both are far from optimal. If the size is stored once per block, a single 4-byte word can be used to keep the size for hundreds of objects, saving considerable amounts of memory. It does mean that the allocator needs to be able to find block metadata from a pointer, to read the

[3] Intra-process parallelism can be very useful even when multiple verification instances are involved. A 64-core system can easily accommodate 4 verification tasks running on 16 cores each, splitting memory between those 4 tasks as needed. If memory becomes scarce, some of the processes can be suspended and swapped out to disk and later, when other tasks have finished, resumed again.

object size associated with the pointer. This particular optimisation also cancels out the extra overhead from adaptive, recursive state splitting employed in our compression scheme. For root and internal nodes, the size of the node (obtained through the allocator) can be used to easily compute the number of children. Likewise, the size of a particular leaf fragment can be cheaply extracted from the allocator metadata.

Second, there are two main classes of objects during state space exploration: the first class contains state vectors that are part of the closed set, and will be reclaimed at the end of the verification run, but not earlier. The second class contains newly generated successor states that may or may not be duplicates of states in the closed set – some of those will go on to be added to the closed set (which may require their re-allocation if compression is enabled) while others will be deallocated when they are found to be duplicates. In other words, some objects are short-lived, and some are very long lived – however, there are few, if any, "in-between" objects. This split would favour a generational allocator – especially since we often know ahead of time whether a particular object will be short- or long-lived (at least in the case where compression comes into play – in other circumstances, the distinction is less clearly cut).

Since compression is such an important ingredient, its requirements need to be considered in the design of a good memory allocator. The considerations laid out above lead to a design where memory is allocated in blocks of same-sized objects. For a number of reasons, it is impractical to reclaim blocks that have been already claimed for a particular object size for another object size (here, parallel access is the main reason that an efficient solution is not known to exist). However, when compression is in use, the state vectors that are allocated during successor generation (into the open set) only exist for a very short time, since they are immediately moved into the compressed state store. Consequently, if the same allocator was to be used for those ephemeral state vectors, a substantial amount of memory would be claimed but unused. While the amount of memory so wasted is only proportional to the number of different state vector sizes (and as such not very large), it can add up to many megabytes. More importantly, this overhead appears in each thread separately and is therefore also proportional to the number of execution threads. So while raw speed is not affected much by a generational approach, memory efficiency can be jeopardised. With those considerations in mind, when state compression is enabled, ephemeral memory is obtained from a simple, special-purpose allocator.

3.2 Pointer Representation

There are two basic options on how to represent pointers: either use raw machine pointers, or use an indirection scheme. The former has a clear advantage in terms of access speed: dereferencing a raw machine pointer is as fast as it gets – any other representation will incur additional costs. On the other hand, most contemporary platforms use pointers that are 64 bits wide – for realistic memory sizes, this constitutes substantial overhead. Current CPUs can physically address at most 48-bit memory addresses, while the rest of the pointer representation is unused – that is 16 bits of memory lost for every pointer. Moreover, there are plenty of places in DIVINE where extra bits packed inside pointers can save considerable amount of memory: the hash tables, for example, can use (some of) those 16 bits to store a small part of the hash value to avoid full object comparisons and speed up lookups at no extra memory expense. Quite importantly, the compression algorithm can use a few of those bits for type-tagging pointers, making it free, in terms of memory use, to distinguish state vector fragments from state component vectors (cf. Section 2).

Moreover, a custom pointer representation enables the allocator to easily find the block header for any given pointer, making it possible to obtain object sizes from pointers to those objects. As explained in previous section, this can save considerable memory in some cases.

The main downside is that the pointer dereference operation needs to consult a lookup table to reconstruct the raw machine pointer. The lookup tables can be represented in such a way that this can be implemented using a single addition instruction, followed by a memory fetch from the lookup table, followed by another addition instruction. Since the lookup tables are relatively small, we can hope that they will always be readily available from fast CPU cache. Maybe more importantly, there will only be very few very hot cache lines in those lookup tables. In our informal testing, the slowdown from this indirection was in single-digits percent range, while the memory savings were quite substantial. Based on this, we have decided to use indirect pointers for storing states and state fragments.

3.3 Implementation

The considerations laid out in previous sections give us a fairly good guidance on how to implement an efficient allocator for use in DIVINE. Our implementation uses a custom pointer type, which is translated to machine pointers on demand, at the cost of an extra memory fetch (which is expected to be served from cache, since the indirection table is usually very hot) and a couple of addition instructions. All data structures in the hot paths of the allocator (object allocation and deallocation) are thread-local and expensive thread synchronisation only happens in special circumstances, usually after some threshold is exceeded: either per-thread freelists have grown too big, or they have become empty; or when all freelists are empty and no pre-allocated memory is available, in which case it needs to be obtained from the operating system.

The shared data structures: indirection tables and lists of shared freelist, are implemented as standard lock-free data structures. Since they are only accessed comparatively rarely, no special precautions need to be taken to make access to them more efficient – the indirection table is almost entirely read-only – it is only written when a new block is allocated. Additionally, a shared counter is maintained to assign blocks to threads (threads claim 16 blocks at once to minimise contention on this counter; the blocks are only allocated when they are needed though).

Table 1. Scaling of pthread_rwlock LLVM model with and without compression and with different splitters.

	W=1		W=2		W=4		W=8	
Configuration	time	scale	time	scale	time	scale	time	scale
no comp.+eph alloc.	7581	1	3785	2.00	1985	3.82	1009	7.51
tree+none+generic	11094	1	6052	1.83	3000	3.70	1499	7.40
tree+old+generic	11625	1	6230	1.87	3074	3.78	1559	7.46
tree+eph+generic	11332	1	5693	1.99	2981	3.80	1523	7.44
tree+eph+hybrid	11258	1	5677	1.98	2973	3.79	1518	7.42
tree+eph+obj-mono	11227	1	5727	1.96	2972	3.78	1519	7.39
tree+eph+obj-rec	11265	1	5743	1.96	3006	3.75	1540	7.31

4 Measurements

We implemented the aforementioned scheme in DIVINE and evaluated it using several large C and C++ models translated into LLVM. We also verified general usability of this scheme by benchmarking a few UPPAAL Timed automata models. All the models can be found in DIVINE source distribution. In this section we will give a detailed analysis of our results.

To measure memory requirements, we used DIVINE's simple statistics output which allows us to track memory allocation during a verification run. We measured resident memory usage, either for DIVINE as a whole or divided by number of states explored; either way, the number in statistics is adjusted by subtracting resident memory used before the model is loaded and before the verification algorithm starts – this allows us to easily compare numbers between different configurations of DIVINE, but still includes all the overheads of the algorithm, such as overhead of thread-local data in a multi-threaded setting. Memory measurements were performed on several computers in a way no memory swapping could have occurred.

For time measurements, we take wall time from DIVINE's report. This time includes the initialisation of the algorithm and the time required to load the model. Time measurements were performed on server with two Intel Xeon E5-2630v2 CPUs at 2.60GHz with 128GB of memory.

Besides the detailed measurements presented in the following sections, we have also measured (using the same set of models) that on average, verification with compression generates states at 77% of the speed of uncompressed algorithm in case of single threaded run, and 73% for 8 workers. We have also measured the scaling behaviour of various configurations of compression and memory allocation schemes. The results of those measurements are summarised in Table 1.

4.1 Allocation schemes

Table 2 shows how memory requirements of DIVINE with tree compression vary based on the allocation scheme used and the number of worker threads. In this case we have considered three variations of allocation scheme:

Table 2. Memory use of LLVM models with compression depending on memory allocator and number of workers.

Name	Average state memory (B)														
	W=1			W=2			W=4			W=8			W=16		
	n/a	old	eph	n/a	old	eph	n/a	old	eph	n/a	old	eph	n/a	old	eph
pt_rwlock	105	90	**88**	106	93	**89**	106	96	**90**	106	104	**90**	109	121	**94**
pt_barrier	60	**45**	**45**	65	**53**	**53**	64	53	**52**	63	54	**52**	63	**53**	**53**
collision	252	232	**229**	253	237	**229**	253	245	**229**	257	261	**235**	265	296	**246**
elevator2	105	**81**	**81**	106	**82**	**82**	106	**82**	**82**	106	**82**	**82**	107	84	**83**
lead-uni_basic	55	**45**	**45**	56	47	**45**	55	48	**45**	56	52	**46**	57	59	**48**
lead-uni_peterson	66	57	**56**	67	59	**56**	67	61	**56**	67	67	**58**	69	79	**60**
hashset-2-4-2	243	202	**191**	244	213	**191**	244	232	**192**	246	270	**194**	250	340	**198**
	W=40														
hashset-3-1	67	77	**47**												

n/a direct allocator, which uses raw machine pointers, and allocates them using general purpose allocator (TBB `malloc`); this scheme stores the size of each entry directly in the memory of the entry, which increases its overhead;
old indirection allocator from Section 3.3 without ephemeral memory optimisation;
eph indirection allocator from Section 3.3 with ephemeral memory optimisation.

It can be clearly seen that indirection allocator with ephemeral memory optimisation is the best option, providing best memory efficiency among the considered options. While the indirection allocator without ephemeral memory optimisation provides comparable efficiency in single-threaded verification, it quickly loses to the optimised version as number of workers increase; this is caused by thread-local overhead of the allocator when allocating short-lived blocks of different sizes. Furthermore, for sufficient number of workers, overhead of the per-thread structures of this allocator can outweigh per-state overheads of the naive solution. These measurements show the importance of an efficient memory allocation scheme for multi-threaded verification, which was further emphasised on `hashset-3-1` model with 630 millions of states, which was verified using 40 worker threads: here, the naive solution has 43 % overhead over our allocator with ephemeral storage, while the allocator without ephemeral storage has 64 % overhead over ephemeral storage allocator. This shows that efficient parallel allocator is a necessary part of memory-efficient parallel verification.

4.2 Compression efficiency

Tables 3 and 4 list overall memory usage and memory usage per state, respectively, including memory usage for various state-vector splitting strategies:

none Verification without compression. For large models (where more than 320 GB RAM was required to finish verification) this value is a lower bound based on average state size and the number of states as reported by a run

Table 3. Total resident memory used for LLVM models, without and with compression with different splitters.

Name	# of	memory usage (GB)					ratio	
			compression					
	states	none	generic	hybrid	obj-mono	obj-rec	best	worst
pt_rwlock	10.7 M	67.9	**0.88**	0.93	0.92	0.94	77.2	72.2
pt_barrier	128.5 M	> 825.4	**5.48**	9.00	8.98	9.27	150.5	89.0
collision	3.0 M	47.6	0.64	**0.63**	0.64	0.64	75.3	74.1
elevator2	33.0 M	> 342.8	2.50	1.93	**1.90**	1.90	180.3	137.4
lead-uni_basic	19.2 M	232.0	**0.81**	1.30	1.30	1.30	288.1	178.3
lead-uni_peterson	12.2 M	146.4	**0.64**	1.03	1.03	1.03	229.6	142.2
hashset-2-4-2	6.7 M	133.3	1.20	**1.15**	1.15	1.16	116.1	111.1
hashset-3-1	626.9 M	> 15109.8	**27.51**	31.96	31.55	31.44	549.1	472.7

Table 4. Total resident memory used for LLVM models, without and with compression with different splitters.

Name	# of	average state memory (B)					ratio	
			compression					
	states	none	generic	hybrid	obj-mono	obj-rec	best	worst
pt_rwlock	10.7 M	6807	**88**	92	91	94	77.2	72.2
pt_barrier	128.5 M	> 6900	**45**	75	75	77	150.5	89.0
collision	3.0 M	17119	229	**227**	231	229	75.3	74.1
elevator2	33.0 M	> 11130	81	62	**61**	61	180.3	137.4
lead-uni_basic	19.2 M	12966	**45**	72	72	72	288.1	178.3
lead-uni_peterson	12.1 M	12926	**56**	90	90	90	229.6	142.2
hashset-2-4-2	6.7 M	21283	191	**183**	184	184	116.1	111.1
hashset-3-1	626.9 M	> 25879	**47**	54	54	53	549.1	472.7

with compression. This bound therefore does not include any overheads of the verification algorithm.

generic Compression with a generic splitter which decomposes a state vector into a balanced binary tree with fixed-sized leaves.

hybrid Compression with a splitter that decomposes a state vector according to the top-level structure of the state vector. The splitter is aware of global symbols, heap, and thread stacks. These chunks are further split in a generic way.

obj-mono An extension of the hybrid approach which further decomposes the state vector, respecting boundaries of smaller objects (individual variables, stack frames and so on). This splitter does not decompose any large individual objects.

obj-rec An extension of the *obj-mono* approach that also allow for decomposition of large objects (> 40 bytes) in a binary fashion.

From the aforementioned tables, the following conclusions can be drawn: tree compression offers excellent savings for LLVM models, providing up to several orders of magnitude decrease in memory requirements. This enables verification

Table 5. Total resident memory used for Timed Automata models, without and with compression.

Name	# of	memory usage (GB)			average state memory (B)			ratio	
		compression			compression				
	states	none	custom	generic	none	custom	generic	best	worst
fischer9_ltsm	0.56 M	0.86	**0.11**	0.13	1656	**212**	249	7.8	6.6
fischer9	0.56 M	0.86	**0.11**	0.13	1656	**211**	249	7.8	6.6
fischer10	2.5 M	4.40	**0.26**	**0.26**	1892	**113**	**113**	16.6	16.6
fischer11	11.1 M	23.2	**1.15**	1.40	2243	**110**	135	20.2	16.6
fischer12	48.8 M	> 119	**4.23**	**4.23**	> 2618	**93**	**93**	28.0	28.0
train-gate9	6.5 M	3.26	**0.91**	1.03	535	**149**	169	3.6	3.2
train-gate10	65.4 M	36.8	**5.94**	11.14	604	**97**	182	6.2	3.3

of models which would be otherwise intractable on any realistic hardware[4]. Furthermore, with the exception of `hashset-3-1`, all of the measured compressed state-spaces can be efficiently verified using a high-end laptop. This is a significant improvement over a dedicated multi-socket computer for verification of the same models that would be needed otherwise (without compression).

Even more significant is the observation that memory requirements per state decrease as the number of states increases, and that they seem to converge to approximately the same number independent of state vector size: even though `hashset-3-1` has almost 4 times larger state vector then `pthread_barrier`, its states are compressed into almost the same size.

Finally, we observe that the effect of advanced splitting algorithms on memory efficiency is mostly negative for LLVM models, even though the achieved compression ratios are still very good in those cases.

Table 5 shows compression results for UPPAAL Timed automata models, using a custom and a generic state vector splitter. The generic version is modelling-language-agnostic and therefore the same as in case of LLVM models. The custom splitter uses a technique similar to the hybrid approach in LLVM. For UPPAAL models, the achieved compression ratios are much lower, but still a significant reduction is obtained. Furthermore, we can see that in this case a custom splitter can significantly improve compression ratio.

5 Conclusions

We have presented a scheme for compressing state vectors in an explicit-state model checker geared towards verification of C and C++ programs. The main contribution of our work is a very efficient scheme for allocating memory and its novel combination with a tree compression scheme. Our approach builds on

[4] If we extrapolate from the biggest model, `hashset-3-1`, we can estimate maximum tractable state space size to be over 40 billion vertices considering high-end server with 2TB of RAM, this could result in around 950 TB of raw uncompressed state space.

earlier solutions but mitigates many of their limitations. The presented scheme is very flexible and offers excellent compression ratios (up to 500×) at a very modest performance penalty. Our tool, building on the presented approach, is realistically capable of exploring on the order of tens of billions of states using commercial, off-the-shelf hardware. Moreover, this number discounts the savings from τ+reduction which alone offers a 50-1000× saving (depending on the model, larger state spaces usually benefit more), together approaching the equivalent of 10^{12} unreduced, uncompressed states (or, considering an average state size of 12 kilobytes, the equivalent of 10000 terabytes of memory).

This represents a considerable improvement in our ability to verify real-world code. With the addition of sufficient parallelism into the mix, very realistic programs can be model-checked in reasonable time and memory using explicit-state techniques. Just as importantly, those advances benefit not only verification of big problem instances on big hardware, but also considerably expands what can be verified using your laptop. In the course of development of DIVINE itself, we increasingly rely on model checking the source code of its components to ensure their correctness. We are quite happy to report that this approach to software development is quickly becoming viable.

References

1. J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.
2. S. Blom, B. Lisser, J. van de Pol, and M. Weber. A Database Approach to Distributed State Space Generation. *Electronic Notes in Theoretical Computer Science*, 198(1):17–32, 2008.
3. J. Geldenhuys, P. de Villiers, and J. Rushby. Runtime Efficient State Compaction in SPIN. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *LNCS*, pages 12–21. Springer, 1999.
4. G. J. Holzmann. State Compression in SPIN: Recursive Indexing And Compression Training Runs. In *The International SPIN Workshop*, 1997.
5. G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage Preserving Reduction Strategies for Reachability Analysis. In *PSTV*, pages 349–363, 1992.
6. A. Laarman, J. van de Pol, and M. Weber. Parallel Recursive State Compression for Free. In *SPIN*, pages 38–56, 2011.
7. D. Peled. Ten Years of Partial Order Reduction. In *Proceedings of the 10th International Conference on Computer Aided Verification*, pages 17–28. Springer-Verlag, 1998.
8. P. Ročkai, J. Barnat, and L. Brim. Improved State Space Reductions for LTL Model Checking of C & C++ Programs. In *NASA Formal Methods (NFM 2013)*, volume 7871 of *LNCS*, pages 1–15. Springer, 2013.
9. V. Štill. State Space Compression for the DiVinE Model Checker, 2013. Bachelor's thesis, Faculty of Informatics, Masaryk University Brno.
10. J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, May 1977.

Weak Memory Models as LLVM-to-LLVM Transformations[*]

Vladimír Štill, Petr Ročkai[**], and Jiří Barnat

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

Abstract. Data races are among the most difficult software bugs to discover. They arise from multiple threads accessing the same memory location, a situation which is often hard to discern from source code alone. Detection of such bugs is further complicated by individual CPUs' use of relaxed memory models. As a matter of fact, proving absence of data races is a typical task for automated formal verification. In this paper, we present a new approach for verification of multi-threaded C and C++ programs under weakened memory models (using store buffer emulation), using an unmodified model checker that assumes Sequential Consistency. In our workflow, a C or C++ program is translated into LLVM bitcode, which is then automatically extended with store buffer emulation. After this transformation, the extended LLVM bitcode is model-checked against safety and/or liveness properties with our explicit-state model checker DIVINE.

1 Introduction

Finding concurrency-related errors, such as deadlocks, livelocks and data races and their various consequences, is extremely hard – the standard testing approach does not allow the user to control the precise timing of interleaved operations. As a result, some concurrency bugs that occur under a specific interleaving of threads may remain undetected even after a substantial period of testing. To remedy this weakness of testing, formal verification methods, explicit-state model checking in particular, can be of extreme help.

Concurrent access to shared memory locations is subject to the so called memory model of the specific CPU in use. Generally speaking, in relaxed memory models, the visibility of an update to a shared memory variable may be postponed or even reordered with other updates to different memory locations. Unfortunately, most programming and modelling languages were designed to

[*] This work has been partially supported by the Czech Science Foundation grant No. 15-08772S.

[**] The contribution of Petr Ročkai has been partially supported by Red Hat, Inc.

[†] The final publication is available at Springer via `http://dx.doi.org/10.1007/978-3-319-29817-7_13`

merely mimic the principles of the underlying sequential computation machine, and therefore lack the syntactic and semantic constructs required to express low-level details of the concurrent computation and the memory model of the underlying hardware architecture in particular. Moreover, for obvious reasons, programmers design parallel algorithms with the *Sequential Consistency* [14] memory model in mind, under which any write to or read from a shared variable is instantaneous and immediately visible to all concurrent threads or processes – an assumption that is far from the reality of contemporary processors.

To protect from inconsistencies due to the reordered or delayed memory writes in the relaxed memory model architectures, specific low-level hardware mechanisms, such as memory barriers, have to be used. A memory barrier makes sure that all the changes done prior the barrier instruction are visible to all other processes before any other instruction *after* the barrier is executed. For more details on how memory barriers work we kindly refer the reader to technical literature. Naturally, the implementation details of a specific relaxed memory model depend on the brand and model of a CPU in use [19].

As a result, programs written in programming languages such as C do not contain enough information for the compiler to emit the code whose behaviour is both correct with respect to the incomplete specification given by the source code and at the same time as efficient as possible. A widely accepted compromise is that sequential code is guaranteed to be semantically correct, but any concurrent data access is the responsibility of the programmer. Such access needs to be guarded with various programming and modelling language addons such as builtin compiler functions, operating system calls, atomic variables with (optional) explicit memory ordering specification, or other non-language mechanisms. Since the correctness of behaviour depends on a human decision, often the resulting binary code does not do exactly what it was intended to do by its developer.

This is exactly where formal verification by model checking can help. The model checking procedure [7] systematically explores all configurations (states) of a program under analysis to discover any erroneous or unwanted behaviour of the program. The procedure can easily reveal states of the program that are only reachable under a very specific thread interleaving; clearly, such states may be very hard to reach with testing alone. Examples of explicit-state model checkers include SPIN [10], DIVINE [4], or LTSmin [12]. Unfortunately, none of the mentioned model checkers have direct support for model checking programs under relaxed memory models. Instead, should a user be interested in verification of a program under relaxed memory model, the program has to be manually (or semi-manually) augmented to capture relaxed memory behaviour.

The main contribution of our paper is in a new strategy to automate model checking of C and C++ programs under relaxed memory model without the need of modification of the interpreter used by the model checker itself. Note that interpreting C and C++ alone is a challenging task and any extension of the interpreter towards relaxed memory models would only make it harder. In fact model checkers do not typically rely on direct interpretation of C or C++ code, but use some other, syntactically simpler, representation of the original program. The model checker DIVINE, for example, interprets LLVM bitcode, which is an intermediate representation of the program created by an LLVM-based compiler.

In order to perform verification of C and C++ programs under relaxed memory model, we suggest to augment the original program and extend it with further data structures (store buffers and a cleanup thread) to simulate the behaviour of the original program under relaxed memory model. However, for the same reasons as above, we avoid direct transformation of C or C++ programs – it would require to parse the complex syntax of a high-level programming language. Instead, we apply the transformation at the level of LLVM bitcode, after the original program is translated by a C++ compiler, but before the representation is passed to the model checker for verification. This scenario allows us to completely separate the weak memory extension from the use of a model checker, hence, it allows us to use any model checker capable of processing LLVM bitcode under Sequential Consistency. Our LLVM bitcode to LLVM bitcode transformation adds store buffer data emulation to under-approximate Total Store Order (TSO) – a particular theoretical model of a relaxed memory model. The transformation is implemented within the tool called LART (LLVM Abstraction and Refinement Tool, Section 7.1 in [22]) that is distributed as a part of DIVINE model checker bundle, under the 2-clause BSD licence.

The rest of the paper is organised as follows. Section 2 lists the most relevant related work, Section 3 gives all the details of the LLVM transformation, Section 4 describes some relevant but rather technical implementation details, Section 5 gives details on an experimental evaluation of our approach, and finally Section 6 concludes the paper.

2 Related Work

The idea of using model checkers to verify programs under relaxed memory models has been discussed first in connection with the explicit-state model checker Murφ [8]. The tool was used to generate all possible outcomes of small, assembly language, multiprocessor programs using a given memory model [21]. This was achieved by encoding the memory model and program under analysis in the Murφ description language, which is an idea applied in many later papers, including this one.

To cope with the rather complex situation around memory models, theoretical models have been introduced to cover as many instances of different relaxed memory behaviours as possible. The currently most used theoretical models are the *Total Store Order* (TSO) [25], *Partial Store Order* (PSO) [25] and *x86-TSO* which is a Total Store Order enriched with interlocking instructions [16]. In those theoretical models, an update may be deferred for an infinite amount of time. Therefore, even a finite state program that is instrumented with a possibly infinite delay of an update may exhibit an infinite state space. It has been proven that for such an instrumented program, the problem of reachability of a particular system configuration is decidable, but the problem of repeated reachability of a given system configuration is not [2].

A particular technique that incorporates TSO-style store buffers into the model and uses finite automata to represent the possibly infinite set of possible contents of these buffers has been introduced in [16]. Since the state space explosion problem is even worse with TSO buffers incorporated into the model, authors of [16] extended their approach with a partial-order reduction technique later on [17].

A different approach has been taken in [11], where the algorithm to be analysed was transformed into a form where the statements of the algorithm could be reordered according to a particular weak memory ordering. The transformed algorithm was then analysed using a model-checking tool, SPIN in that case.

A lot of research has been conducted to actually detect deviation of an execution of the program on a relaxed memory model architecture from an execution under Sequential Consistency (SC). An SC deviation run-time monitor using operational semantics [18] of TSO and PSO was introduced in [6], where authors considered a concrete, sequentially consistent execution of the program, and simulated it on the operational model of TSO and PSO by buffering stores, as long as they generated the same trace as the SC execution. Another approach to detect discrepancies between a sequential consistency execution and real executions relied on axiomatic definition of memory models and (SAT-based) bounded model checking [5].

The problem of relaxed memory model computation has been addressed also in the program analysis community. Given a finite-state program, a safety specification and a description of the memory model, the framework introduced in [20] computes a set of ordering constraints that guarantee the correctness of the program under the memory model. The computed constraints are maximally permissive: removing any constraint from the solution would permit an execution that violates the specification. To address the undecidability of the problem, an abstraction from precise memory models has been considered by the BLENDER tool [13]. The tool employs abstract interpretation to deliver an effective verification procedure for programs running under relaxed memory models.

Another program analysis tool, called OFFENCE, was introduced to ensure program stability [1] by inserting a memory barrier instruction where needed – an approach also used in [17]. The problem of relaxed memory model and correct placement of synchronisation primitives is also relevant for the compiler community [9].

The problem of LTL model checking for an under-approximated TSO memory model using store buffers was also evaluated in [3], where authors proposed transformation of the DVE modelling language programs to simulate TSO.

3 Emulation of Relaxed Memory in LLVM Bitcode

We have chosen to provide an under-approximation of the TSO memory model, both for its simplicity and the fact that it closely resembles the memory model

```
    int x = 0, y = 0;

1   void thread0() {                      1  void thread1() {
2       y = 1;                            2      x = 1;
3       cout << "x = " << x << endl;      3      cout << "y = " << y << endl;
4     }                                   4  }
```

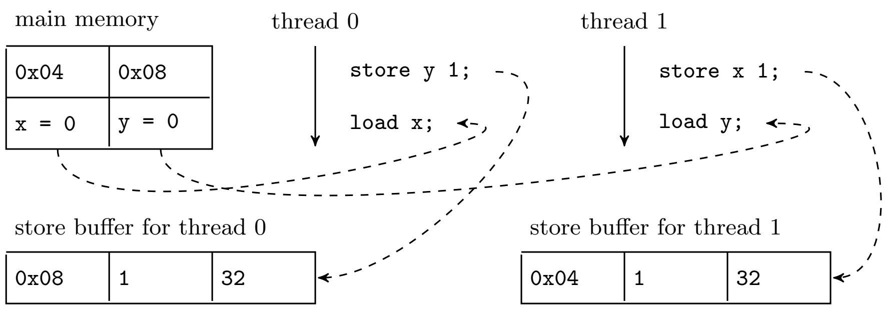


Fig. 1. In this example, each of the threads first writes into a global variable and later reads the variable written by the other thread. Under sequential consistency, the possible outcomes would be $x = 1, y = 1$; $x = 1, y = 0$; and $x = 0, y = 1$, since at least one write must proceed before the first read proceeds. However, under TSO $x = 0, y = 0$ is also possible: this corresponds to the reordering of the load on line 3 before the independent store on line 2, and can be simulated by performing the store on line 2 into a store buffer. The diagram shows (shortened) execution of the listed code. Dashed lines represent where given value is read from/stored to.

used by x86 computers. In this memory model, all stores are required to become visible in the same order as they are executed; however, loads can be executed before independent stores. This situation can be emulated by per-thread store buffers – stores are performed into store buffers and later flushed into main memory. Loads then have to first consult their thread's respective store buffer, and if it does not contain the address in question, proceed by consulting the main memory. Loads do not see changes that are recorded only in store buffers of other threads. We can see an illustration of the TSO memory model, and its simulation using store buffers, in Figure 1. While in the sequentially consistent case, the result $x = 0, y = 0$ would not be possible, under TSO it is a valid output of the program, and indeed it can be proved reachable by running DIVINE on the transformed code. Note that store buffers are flushed non-deterministically, using a dedicated thread; in particular, we run a dedicated flushing thread for each worker thread.

Note that we deliberately avoid precise (unbounded store-buffer) simulation of the theoretical TSO memory model, as this could easily result in infinite state space of the program under verification. However, the store buffer size can be passed as a parameter to the bitcode transformation. This way, we can make both reachability and LTL verification decidable and connect it seamlessly to an existing explicit-state framework. Please note that this approach only under-approximates the set of all TSO behaviours. I.e., when DIVINE finds a counterexample in the modified model, this counterexample can indeed occur in some runs of the given program on some real hardware with TSO semantics. On the other hand, not finding a counterexample does not guarantee error free execution on machines with store buffers deeper than specified for verification. Obviously, setting the size of store buffers is a matter of compromise – larger buffers will result in more precise verification, but also in a larger state spaces.

3.1 Infinite Delay Problem

For safety properties, such as assertion violation and/or memory safety, delaying writes indefinitely (never flushing them from a store buffer) is not a problem, as any violation of safety property is witnessed by finite path and for each run with infinite delay, there also exists (possibly finite) run where each write is eventually flushed. In infinite runs, however, such as those constructed as counterexamples to liveness properties, infinite delays could pose a problem. Imagine, for example, the following two threads:

```
bool x = false, y = false;
```

```
void thread0() {
    y = true;
    while ( !x ) { AP( w0 ) }
    for (;;) { /* work */ }
}
```

```
void thread1() {
    x = true;
    while ( !y ) { AP( w1 ) }
    for (;;) { /* work */ }
}
```

and a liveness property written (using LTL) as $FG(\neg w_0 \wedge \neg w_1)$. Assuming a separate thread to perform store buffer flushes, it is easy to see that this property holds only if the buffers are actually flushed on every possible run. However, since flushing happens non-deterministically, it may actually never happen on an infinite run. While this can be viewed as theoretically correct, it does not correspond to any real-world behaviour, where delayed writes will eventually finish and the program eventually proceeds. To counteract this inconsistency, we ask our model checker to assume weak fairness [15], where it is guaranteed that every non-blocking thread has performed infinitely many actions in an infinite run.

In [3], authors proposed to handle this problem by extending LTL specification to include this store buffer fairness criteria. In our case though, we have chosen to implement our transformation in a way which does not require any additional specification and store buffer fairness is implied by the standard weak fairness.

3.2 Invalidated Variable Store Problem

Another issue to deal with are delayed flushes from a store buffer that come at the time when the object that should be written into does not exist anymore in the main memory. As both memory allocation and stack depth can change at the run-time, it might happen that an entry in the store buffer points to invalid location (either given memory chunk was deallocated by the user, or it lived in a stack frame that has already been abandoned). To solve this problem, we would need to make sure that inaccessible addresses are evicted from the store buffers. For dynamic memory, this can be done by overriding the function which deallocates objects from memory in such a way that it first iterates over all store buffers and evict entries into the to-be-freed memory before calling the original deallocate function.

For stack memory, however, the situation is more complicated – it is not sufficient to evict all the stack-frame-allocated memory from store buffers before returning from a function, because an exception can cause stack unwinding, which can also result in invalid references in store buffers. This means that cleanup handlers [24] need to be added to each function to deal with the situation.

4 Implementation

First of all, let us briefly explain how LLVM bitcode is used by our target model checker DIVINE to support for C/C++ verification. There are two levels below the LLVM bitcode of the program to be verified – an interpreter and an LLVM *userspace*. The interpreter is used directly by the model checker to generate and explore the state space graph by executing LLVM instructions. The interpreter detects errors such as invalid memory dereference, memory leaks, assertion violations, etc. The interpreter has to be aware of threads and dynamic memory management, hence, its role is similar to what the CPU and the core of the operating system do when executing the code natively. The userspace, on the other hand, corresponds to the runtime of the programming language, that is, it provides LLVM bitcode for the basic libraries required by the given programming language and/or threading model. The userpsace and interpreter together provide the user with a standards-compliant interface for user's programming language of choice.

While in general, the separation of work between the interpreter and userspace could be almost arbitrary (one could, for example, include the entire pthread library in the interpreter), it is advantageous to keep the interpreter as simple as possible, pushing most of the required functionality into the userspace. Therefore, DIVINE provides a fairly small set of intrinsic functions (sixteen in total), which give access to the necessary functionality provided by the interpreter. The rest is left to userspace.

The support for relaxed memory verification, such as functions that simulate store buffers, thus need not come separately for every program to be verified under relaxed memory model, but may actually become a part of the DIVINE LLVM userspace. However, it is not possible to implement weak memory simulation through addition of userspace functions alone – we need to change the behaviour of memory manipulation instructions (such as loads, stores, and fences). For this reason, we implemented an LLVM to LLVM bitcode transformation pass, which translates relevant instructions into calls to the relevant userspace functions. The actual simulation of the memory model is thus implemented within the userspace and is separate from the original program. As a result of this design choice, this transformation can be easily modified to work with other LLVM model checkers and with different weak memory models.

4.1 Updates to LLVM Userspace

Currently, LLVM userspace provides replacement functions for `load`, `store` and `fence`. The relevant userspace functions can be identified by their `__lart_weakmem` prefix. Store buffers are represented by a thread-local array with one record for each store – this record contains the address, the value itself and the bit width of the value. We have chosen to limit a single store to 64 bits, which is the usual size atomically written by modern CPUs and also the maximal size of standard integer types in C. Each store then pushes a record into the local store buffer, while loads first consult the local store buffer for an up-to-date value, and if it is not present proceed to load from memory. A fence flushes all the entries from the local store buffer.

Note that block memory manipulation functions have to be replaced too, to protect them from bypassing the store buffers. Hence, the userspace provides replacements for block memory manipulation functions such as `llvm.memmove`, `llvm.memcpy`, etc.

Further, atomic LLVM instructions, e.g. `cmpxchg`, are rewritten within the transformation to use only functions implemented within the userspace. However, we currently only support sequentially-consistent ordering of atomics (which is the default ordering for atomic variables in C++11). Further extensions to support all atomic access orderings supported by LLVM/C++11 are planned.

Finally, attention had to be paid to initialisation of the store buffers. Due to the nature of global variable constructors in C++ which can run in arbitrary order, we cannot use non-trivial constructors for store buffers, as this could cause the constructor to run after some calls to `__lart_weakmem_*` functions have already happened. Therefore, the store buffer array is initialised to a null pointer and allocated in the first call to one of the `__lart_weakmem_*` functions.

4.2 LLVM to LLVM Transformation

The transformation is implemented as part of the LART tool. It basically iterates over all the instructions in the original LLVM bitcode and replaces some of them with calls to the corresponding replacement functions.

To perform this transformation correctly, we had to introduced special LLVM function attributes: *bypass*, *tso*, and *sc*, denoting in what mode a particular function should operate. Functions marked *bypass* are not subject to the transformation at all, functions marked *tso* are fully processed by the transformation as indicated above. In functions marked *sc*, additional memory barriers are inserted at the beginning of the function and after a call to any non-SC function.

Note that it is important that the functions which implement the relaxed weak memory model itself are not transformed; for this reason, all `__llvm_weakmem_*` functions are annotated as *bypass*. The default behaviour of the transformation on functions that are not annotated with any of the attributes can be set by a parameter passed to the transformation.

Since LLVM allows loads and stores larger than 64 bits (either large scalar types, such as 128 bit integers, or aggregate values), we first break these large loads and stores into chunks of at most 64 bit-wide operations in a separate transformation pass and only after this is done, we perform the instruction substitution transformation as outlined above.

Finally, to avoid interference from compiler optimisations, some of the memory accesses in our functions had to be marked volatile and we had to prevent inlining of some of the functions (since inlining would discard function attributes). Likewise, all the exposed functions had to be marked `noinline`.

4.3 State Space Reduction

Store buffers substantially increase the size of the state space, hence it is necessary to counteract this growth. DIVINE provides powerful reduction techniques out of the box, based on analysis of instruction visibility. Those reductions are, however, rendered less effective by interactions with the store buffer: in particular, any TSO load or store is treated as visible by the $\tau+$ reduction due to global variable access within the TSO load/store implementation.

Fortunately, it is possible to reduce the overhead of store buffers by entirely bypassing their use for memory locations that are private to a particular thread. However, since the entire logic of TSO stores is handled in the userspace, it is necessary to expose an additional intrinsic (builtin) function in the model checker, which, for a given address, decides whether the address is visible from any other threads.

As far as correctness is concerned, when we realise that from the point of view of the model checker, store buffers are part of the global memory, the argument carries over from the analogical construct (store visibility) used in $\tau+$ reduction [23]. Any pointers currently residing in store buffers – and hence, capable of revealing new memory locations to foreign threads – are treated as global; hence, a delayed write of such a pointer cannot incorrectly hide intervening stores (into locations that were previously thread-private but revealed by the pointer living in a store buffer).

5 Evaluation

We evaluated our approach on a few models, all of which can be found in examples in source distribution of DIVINE[1]. Descriptions of the models used can be found in Table 1. All measurements were performed on a laptop with Intel Core i7-3520M, running at 3.4 GHz, with 8 GB of memory. DIVINE used

[1] online: https://divine.fi.muni.cz/trac/browser/examples/llvm/weakmem/

Table 1. Models used for evaluation

`simple_sc`	Model based on figure 1, SC, asserting that $x = 0, y = 0$.
`simple_mtso`	Same model, but manually modified to use TSO for relevant variables.
`simple_stso`	Same model, workers are auto-transformed to TSO, the rest is SC.
`simple_tso`	Same model, fully transformed to TSO.
`peterson_sc`	Peterson's mutual exclusion algorithm.
`peterson_tso`	The same, automatically transformed to TSO.
`fifo_sc`	First-in, first-out, lockless inter-thread queue, as used in DIVINE.
`fifo_tso`	Automated TSO transform of `fifo_sc` above.

4 threads for verification and never depleted available memory (loss-less state space compression was enabled).

5.1 Results

The results of verification with DIVINE can be seen in Table 2. In all cases, Context-Switch-Directed Reachability [26] was used, as it performed much faster than regular reachability for the TSO simulation case. From the results, we can see significant increase of state space size when store buffers are enabled. This is due to two factors – one of them is that the store buffers themselves increase the state space size, as they can be flushed non-deterministically anywhere between the given store and the nearest memory barrier. The other issue is the interference with $\tau+$ reduction mentioned in Section 4.3. As can be seen in the case of `peterson_sc` and `peterson_tso` with store buffers of size 0 (in this case value is stored into store buffer and immediately flushed out within one transition in the state space), this effect is quite strong.

As for the differences between different versions of the `simple` model, the state space size is clearly dependent on how many of the loads and stores are treated as TSO – in case of full TSO transformation all library functions are also in TSO, therefore state space size is increased far more. The difference between `simple_mtso` and `simple_stso` is more subtle: in the case of `simple_stso` our transformation adds memory barriers into SC functions, at their beginning and after any call to non-SC function. While the second case is rarely present in our model, the first case makes any function call observable, as a flush will be considered observable by $\tau+$ reduction (due to an accesses to the store buffer).

6 Conclusion

We have introduced an LLVM to LLVM transformation that extends a program with relaxed memory simulation and we have shown that such an extended program can be passed to a model checker to perform verification of C/C++ programs under a relaxed memory model. A key attribute of our approach is that no updates to the model checker (which is based on sequential consistency) are needed. The preliminary experiments show the approach as such is feasible,

Table 2. Results of `divine verify` for our examples.

model	store buffer size	assertion violated	# of states	reduced # states	memory [GB]	time [s]
`simple_sc`	N/A	no	205	N/A	0.16	1
`simple_mtso`	1	**yes**	6.89 k	N/A	0.17	3
`simple_stso`	1	**yes**	10.7 k	10.7 k	0.17	6
`simple_tso`	1	**yes**	24.7 M	537.2 k	3.18	20318
`peterson_sc`	N/A	no	1.68 k	N/A	0.16	1
`peterson_tso`	0	no	55.9 k	N/A	0.17	38
`peterson_tso`	2	**yes**	2.86 M	95.7 k	0.79	990
`peterson_tso`	3	**yes**	4.70 M	129.9 k	1.21	1610
`fifo_sc`	0	no	6951	N/A	0.73	20
`fifo_tso`	1	no	–	44 M	–	–

even though the growth of the state space is significant. Finally, the verification of the `fifo_tso` model is, in itself, a valuable result, as the code in question is sensitive to memory ordering and until now we were only able to verify it under the assumption of sequential consistency.

As our future work we intend to improve the implementation and also implement support for weaker memory models, such as Partial Store Order. As a research goal, we want to extend LART to automatically annotate some functions as SC, whenever it can be statically decided that such an annotation has no influence on the verification result, counteracting the growth of the state space. Further improvements of reductions supported by DIVINE and their interaction with store buffer simulation, and thread-local memory in general, could also significantly reduce the state space.

References

1. J. Alglave and L. Maranget. Stability in weak memory models. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 50–66, Berlin, Heidelberg, 2011. Springer.
2. M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 7–18, New York, NY, USA, 2010. ACM.
3. J. Barnat, L. Brim, and V. Havel. LTL Model Checking of Parallel Programs with Under-Approximated TSO Memory Model. In *Application of Concurrency to System Design (ACSD)*, pages 51–59. IEEE, 2013.
4. J. Barnat, L. Brim, V. Havel, and J. Havlíček et al. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *CAV*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.
5. S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV*, volume 5123 of *LNCS*, pages 107–120. Springer, 2008.
6. J. Burnim, K. Sen, and C. Stergiou. Sound and Complete Monitoring of Sequential Consistency in Relaxed Memory Models. Technical Report UCB/EECS-2010-31, EECS Department, University of California, Berkeley, March 2010.

7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT press, 1999.
8. D. Dill. The Murphi Verification System. In *Computer Aided Verification*, volume 1102 of *LLNC*, pages 390–393. Springer, 1996.
9. X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *International Conference on Supercomputing (ICS'03)*, pages 285–294. ACM, 2003.
10. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004.
11. B. Jonsson. State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). *SIGARCH Comput. Archit. News*, 36:65–71, June 2009.
12. G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *LNCS*, pages 692–707. Springer, 2015.
13. M. Kuperstein, M. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *Programming language design and implementation (PLDI'11)*, pages 187–198. ACM, 2011.
14. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.
15. D. J. Lehmann, A. Pnueli, and J. Stavi. Impartiality, Justice and Fairness: The Ethics of Concurrent Termination. In *Automata, Languages and Programming (ICALP)*, volume 115 of *LNCS*, pages 264–277. Springer, 1981.
16. A. Linden and P. Wolper. An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models. In *Model Checking Software*, volume 6349 of *LNCS*, pages 212–226. Springer, 2010.
17. A. Linden and P. Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *Proceedings of the 18th international SPIN conference on Model checking software*, pages 144–160, Berlin, Heidelberg, 2011. Springer.
18. S. Mador-Haim, R. Alur, and M. M. K. Martin. Specifying relaxed memory models for state exploration tools. In $(EC)^2$*: Workshop on Exploting Concurrency Eficiently and Correctly*, 2009.
19. P. E. Mckenney. Memory Barriers: a Hardware View for Software Hackers, 2009.
20. M.Kuperstein, M. T. Vechev, and E. Yahav. Automatic inference of memory fences. In *Formal Methods in Computer-Aided Design*, pages 111–119. IEEE, 2010.
21. S. Park and D. Dill. An executable specification and verifier for relaxed memory order. *IEEE Trans. on Computers*, 48(2):227–235, 1999.
22. P. Ročkai. *Model Checking Software.* Disertation thesis, Masaryk University, Faculty of Informatics, 2015.
23. P. Ročkai, J. Barnat, and L. Brim. Improved State Space Reductions for LTL Model Checking of C & C++ Programs. In *NFM*, volume 7871 of *LNCS*, pages 1–15. Springer, 2013.
24. P. Ročkai, J. Barnat, and L. Brim. Model Checking C++ with Exceptions. *Automated Verification of Critical Systems*, 70, 2014.
25. CORPORATE SPARC International, Inc. *The SPARC architecture manual (version 9).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
26. V. Štill, P. Ročkai, and J. Barnat. Context-Switch-Directed Verification in DIVINE. In *Mathematical and Engineering Methods in Computer Science MEMICS 2014*, volume 8934 of *LNCS*, pages 135–146. Springer, 2014.

Using Off-the-Shelf Exception Support Components in C++ Verification

Vladimír Štill
Faculty of Informatics,
Masaryk University
Brno, Czech Republic
Email: xstill@fi.muni.cz

Petr Ročkai
Faculty of Informatics,
Masaryk University
Brno, Czech Republic
Email: xrockai@fi.muni.cz

Jiří Barnat
Faculty of Informatics,
Masaryk University
Brno, Czech Republic
Email: barnat@fi.muni.cz

Abstract—**An important step toward adoption of formal methods in software development is support for mainstream programming languages. Unfortunately, these languages are often rather complex and come with substantial standard libraries. However, by choosing a suitable intermediate language, most of the complexity can be delegated to existing execution-oriented (as opposed to verification-oriented) compiler frontends and standard library implementations. In this paper, we describe how support for C++ exceptions can take advantage of the same principle. Our work is based on DiVM, an LLVM-derived, verification-friendly intermediate language.**

Our implementation consists of 2 parts: an implementation of the `libunwind` platform API which is linked to the program under test and consists of 9 C functions. The other part is a preprocessor for LLVM bitcode which prepares exception-related metadata and replaces associated special-purpose LLVM instructions.

Index Terms—**Model Checking, Exceptions, C++, Unwinder**

I. Introduction

Today, formal verification methods are not commonly used in software development, even though they are superior to traditional testing approaches in many respects. One particular example is model checking, which can be used to control non-determinism in programs, especially when it arises from parallelism. Formal methods can also be used to extend testing coverage (e.g. via systematic fault injection), verification of liveness properties or verification of global safety properties (such as global assertions). However, to make those advantages actually available to software developers, verification tools must be easy to integrate into existing workflows. If the use of verification tools requires substantial effort (as compared to testing), the costs associated with formal methods can outweigh the savings they provide. This is especially true for modern development processes (especially in commodity software), where there is little time for a separate modeling and design phases.

This work has been partially supported by the Czech Science Foundation grant No. 15-08772S and by Red Hat, Inc.

For this reason, both the academic and industrial communities [2] increasingly seek to develop and use tools which work with mainstream programming languages. However, support for these programming languages – especially when compared to special-purpose modelling formalisms – brings new complexity to verification tools. Programs written in such languages are usually more complex and on a lower level of abstraction than models specifically built for analysis tools. Additionally, many programming languages contain features with no counterparts in a typical modeling language, such as dynamic memory, run-time type information and introspection (RTTI), exception handling, or template instantiation. Moreover, programs written in these languages usually make use of extensive standard libraries. Therefore, the verifier either has to include all of the language and library functionality as primitives, or it has to provide an implementation which is added to the verified program just like a traditional library.

The paper is structured as follows: the remainder of Section I gives motivation, context and contribution of this work. Section II describes the mechanisms that C++ implementations typically use in order to support exceptions. The following Section III then details how LLVM is interpreted in DIVINE 4, in particular the parts relevant to exception handling, such as the stack layout. Section IV and Section V discuss the new components: the LLVM transformation and the unwinder, respectively. Section VI surveys the related work and finally, in Section VII, we evaluate our approach and we summarise our findings in Section VIII.

A. Motivation

In many cases, it is impractical to re-implement the entire programming language and its support libraries. Verification tools can, however, take advantage of existing compilers or libraries to deal with some of the complexity. For example, verification can be substantially simplified by translating the source code into an intermediate representation (IR) using an existing compiler frontend. If the compiler in question can emit intermediate representation after it has been optimised, the verification result is independent of the correctness of the (complex and error-prone) optimiser: any problems introduced by the optimiser will be caught by the verification tool.

In case of C++, a suitable frontend is the clang compiler, which uses LLVM as its IR. Since LLVM can optimise the IR and produce executable code on many platforms from a single optimised IR file, the verification effort does not need to be repeated for each target platform separately. Of course, the code generator (which is comparatively simple when compared to the platform-neutral optimiser) still needs to preserve the semantics of the program – otherwise, it would invalidate the verification result.

As an alternative to re-using finished, execution-oriented components, one could only support a subset of a programming language (i.e. exclude the parts that are hard to support in a verification tool). However, this weakens the case for supporting mainstream programming languages: it prevents developers from verifying production code. This is especially true for standard libraries, as programming without them requires the programmer to implement everything from scratch. Finally, the standard library is often implemented in the programming language it is part of, and is therefore another good candidate for sharing code with execution-oriented implementations of the language. Unfortunately, upstream implementations of standard libraries usually make extensive use of advanced language features. Consequently, in order to re-use existing standard library implementations, more complete language support is required in the verifier.

Exceptions are among the features that are both widely used (including by the standard library) and tricky to implement. Their use is, however, also common outside of the standard library: libraries like `boost` and application-level code often take advantage of this capability. This is natural, since exceptions simplify error handling and usually require less boilerplate code than any of the alternatives. Furthermore, even though many C++ standard library implementations can be built without exception support[1], this change can significantly affect its behaviour (and as such, validity of the verification result). Finally, error handling paths, including exception propagation, are an important target for analysis by verification tools, as they are both hard to test by more conventional means and likely to contain errors – this naturally arises from the fact that their purpose is to handle unlikely side cases which can be hard to accurately reproduce with testing. A model checker, on the other hand, can take advantage of its built-in support for non-determinism to rigorously explore error paths.[2]

B. Component Re-Use

Unfortunately, off-the-shelf components from execution-oriented language kits do not provide a complete toolbox that would allow verification tool developers to simply concentrate on verification. The difficulties roughly fall into two categories: first, the components interact with each other and with the system for which they were originally designed and second, it is often not at all obvious which components are suitable for re-use and which are not. When a component C is re-used, all the interfaces it uses must be provided as well. There are 3 basic ways in which this can be arranged:

1. re-use another component, D, which provides this interface; this is only possible if all interfaces D uses are already available or can be provided
2. modify component C to avoid its dependency on the interface in question
3. re-implement the interface as a new, possibly tool- or verification-specific component

C. Contribution

The main contribution of this paper is twofold: first, we identify the components that are best re-used and those which are best re-implemented and show that this decision crucially depends on the underlying intermediate language. Second, we provide implementations of the components which cannot be re-used in a form that is easy to integrate into both existing and future verification tools. One of the components works as an LLVM transformation pass, and could be used with any LLVM-based tool. The other component targets the DiVM language [14] specifically, and will therefore only work with tools which understand this language.[3]

The goal of this paper, especially in the context of our previous work on the topic of C++ exceptions in verification [13], is to aid authors of verification tools to minimise costs and effort associated with inclusion of exception support. Depending on the characteristics of the tool, either the approach described in [13] or the one in this paper might be more suitable. Overall, in a verifier which can handle the DiVM language or equivalent, the approach given in this paper is simpler to implement and more robust. A more detailed comparison of the two approaches is given in Section VII-B.

All source code related to this paper, along with more detailed benchmark results and other supplementary material, are available online under a permissive open-source licence.[4]

D. Implementation

Our primary implementation platform is the DIVINE model checker [1]. The C++ support in DIVINE has several components: first, DIVINE uses clang to translate C++ into LLVM IR. As outlined above, the verifier does not need to handle complex syntactical features of C++ this way. A few verification-specific transformations are done on the LLVM IR before it is converted into the DiVM language for execution in DIVINE's Virtual Machine. The VM executes instructions and performs safety checks, such as bound checking. Alone, these components provide basic support for C++. In order to support

[1]There are cases where not using exceptions makes sense: if the end-user code makes no use of them but the standard library is compiled with exception support, the requisite metadata tables only serve to increase the size of the compiled program.

[2]This is a form of fault injection. When using a model checker, it is only necessary to modify the function where the error may arise (e.g. the `malloc` function may be modified to return a `NULL` pointer non-deterministically). The model checker will then take care of exploring all possible combinations of succeeding and failing memory allocations in the program.

[3]DiVM is a relatively small extension of the LLVM IR, therefore extending tools which work with pure LLVM to also support DiVM may be quite easy.

[4]https://divine.fi.muni.cz/2017/exceptions

features such as RTTI and exceptions, it is also necessary to provide a runtime support library and an implementation of the standard library. These libraries in turn rely on a C standard library and on a threading library (`pthreads` on POSIX compatible systems). Those libraries are provided by DiOS, a small, verification-oriented operating system which runs inside DiVM.

As discussed above, building those libraries into the verifier is impractical due to cost and time constraints. There is, however, another important reason why these should be kept out of the verification core: any extension of the verifier increases risks of implementation errors, and the more complex these extensions are, the higher are the associated risks. Moreover, any such errors in the verifier can lead to incorrect verification results. For this reason, DIVINE ships source code implementing these libraries as separate modules; this source code is later compiled into LLVM IR and linked to the verified program. This way, the libraries are subject to the same error checking as user code, and any errors in their implementation that are exposed by the user program will be detected by the verifier.

Additionally, whenever off-the-shelf components are re-used, it is preferable to keep verification-specific changes at minimum. The standard C library in DIVINE is based on PDCLib, a small, portable, public domain C library. The copy of PDCLib in DIVINE includes a few modifications (the C library interfaces directly with the operating system in many cases, therefore it is necessary to port it to work with the verifier, much like it would be necessary to port it to a new operating system). For threading support, DIVINE ships with a custom implementation of the `pthread` library (so far, no existing implementation of the `pthread` interface which could be re-used has been identified). For C++ support, `libc++abi` (the runtime library) and `libc++` (the standard library) are used. Both of these libraries are maintained by the LLVM project and work on many Unix-like systems.

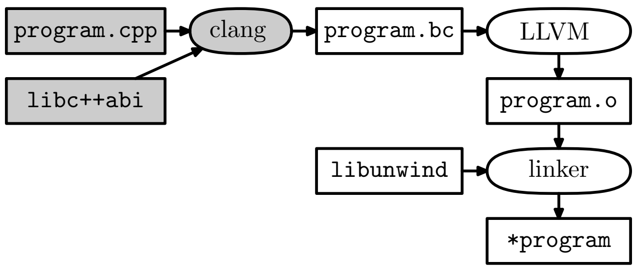


Fig. 1. Components involved in exception support in the standard clang/LLVM stack. Under the scheme proposed in this paper, the highlighted elements are shared between verification and execution environments.

E. Components for Exception Support

Unlike other features of C++, exceptions are neither handled by the standard or runtime libraries alone, nor delegated to the C standard library (as C has no support for exceptions). Instead, `libc++abi` provides exception support with the help of a platform-specific *unwinder library* which is responsible for stack introspection and unwinding (removal of stack frames and transfer of control to exception-handling code). The interaction of these components is illustrated in Figure 1.

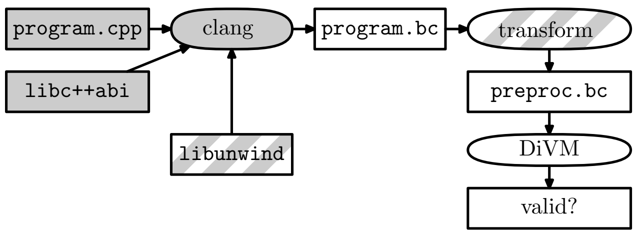


Fig. 2. Components involved in exception support in the DIVINE 4 C++ verification stack. The solid-filled elements are re-used without modification from the execution-oriented clang/LLVM stack (cf. Figure 1). The hatch-filled components are the additions described in this paper.

For this reason, DIVINE has to either provide an unwinder implementation compatible with `libc++abi`, or modify `libc++abi` to use custom code for exception handling. In DIVINE 3, the latter approach was used, as it was deemed easier at the time [13]. However, while basic exception support was easier to achieve this way, the approach also had its disadvantages. First, the LLVM interpreter in DIVINE 3 had special support for exception-related functionality. Second, the `libc++abi` code for exception handling was replaced, which had 2 important consequences: first, the replacement code was not comprehensive enough[5] and second, this meant that the replaced part of `libc++abi` was not taken into account during verification.

In this paper, we instead take the first approach: re-use `libc++abi` in its entirety and provide the interfaces it requires. Therefore, we have implemented the `libunwind` interface used by `libc++abi` for stack unwinding and an LLVM transformation which builds metadata tables that `libc++abi` needs to decide which exceptions should be caught, how they should be handled and which functions on the stack need to perform cleanup actions. The situation is illustrated in Figure 2.

Using the original `libc++abi` code means that all features of the C++ exception system are fully supported and verification results also cover the low-level exception support code. That is, this portion of the code is identical in both the bitcode which is verified and in the natively executing program.[6] Finally, the proposed design is easier to extend to other programming languages.

F. Other Components in Use

In line with the principles outlined so far, the implementation of the C and C++ standard libraries (and the C++ runtime library) used in DIVINE are third party code with only

[5]That is, some of the less frequently used features of C++ exceptions were handled either incorrectly or not at all. That is to say, the size of the `libc++abi` portion that would have needed to be re-implemented was initially underestimated.

[6]Clearly, the `libunwind` implementation is different in those two environments, and therefore correctness of the platform-specific implementation of `libunwind` must be established separately.

minimal modifications. The C standard library implementation (PDCLib) consists of approximately 38 thousand lines of code, while the C++ runtime library (libc++abi) and the C++ standard library (libc++) contain 8 and 12 thousand lines of code, respectively.

Standard libraries inevitably contain platform-specific code, and this is also true of the implementations bundled with DIVINE. The modifications due to the porting effort were, however, quite minimal, since DiOS already provides a very POSIX-like interface. The C library was, unsurprisingly, affected the most: changes in memory allocation, program startup- and exit-related functions and in handling of the errno variable were required. In libc++, however, the changes were limited to platform configuration macros and the only change in libc++abi was a DiOS-specific tweak in allocation of thread-local storage for exception handling.

Since user programs and libraries alike rely on the POSIX threading API (also known as pthread), this API is provided by DiOS and is implemented in about 2000 lines of C++. The libunwind implementation introduced in this paper brings in additional 350 lines of code (the implementation is done in exception-free C++). Likewise, the C library and everything above also depends on low-level filesystem access routines provided by the operating system. In DiOS, this IO and filesystem layer (VFS[7]) is implemented in about 5500 lines of C++ code and uses exceptions heavily for error propagation.

So far, all the components mentioned in this section are linked with the user program to form the final bitcode file for verification. For comparison, the verification core of DIVINE (the DiVM evaluator, memory management and the verification algorithm), amounts to roughly 6 thousand lines of C++. Finally, there is about 2500 lines of code which implement various transformations on the LLVM bitcode. Out of these 2500 lines, less than 300 are part of the exception-related extension described in this paper.

II. Exceptions in C++

Throwing an exception requires removal of all the stack frames[8] between the throwing and catching function from the stack (*unwinding*). Therefore, exception handling is closely tied to the particular platform and is described by ABI[9] for the platform. Commonly, exception handling is split into two parts, one which is tied to the platform (the *unwinder library* which handles stack unwinding) and one which is tied to the language and provided by the language's runtime library.[10] These two parts cooperate in order to provide exception handling for a given language; however, this communication is not standardised in any cross-platform fashion. For this reason, we will now focus on zero-cost exceptions based on the Itanium ABI, an approach which is used across various Unix-like systems on x86 and x86_64 processor architectures and is the preferred basis for LLVM exceptions. Nevertheless, it is possible to generalize our results to other implementations.

A. Zero-Cost Exceptions

The so-called zero-cost exceptions are designed to incur no overhead during normal execution, at the expense of relatively costly mechanism for throwing exceptions. This in particular means that no checkpointing is possible. Instead, when an exception is thrown, the exception support library, with the help of *unwind tables*, finds an appropriate *handler* for the exception and uses the *unwinder* to manipulate the stack so that this handler can be executed. The search for the handler is driven by a *personality function*, which is provided by the implementation of the particular programming language.

The personality function is responsible for deciding which handler should execute (the handler selection can be complex and language-specific). In general, there are two types of handlers, *cleanup handlers*, which are used to clean up lexically scoped variables (and call their destructors, as appropriate) and *catch handlers*, which contain dedicated exception-handling code. The latter typically arise from catch blocks. Another major difference between those two types of handlers is that catch handlers stop the propagation of the exception, while cleanup handlers let propagation continue after the cleanup is performed. While cleanup handlers are usually run unconditionally, the catch handler to be executed, if any, is determined by the personality function.[11] In C++, the personality function selects the closest catch statement which matches the thrown type (the match is determined dynamically, using RTTI). The personality function consults the unwind tables, in particular their *language-specific data area (LSDA)*, to find information about the relevant catch handlers.

When an exception is thrown, the runtime library of the language creates an *exception object* and passes it to the unwinder library. The actual stack unwinding is, on platforms which build on the Itanium ABI, performed in two phases. First, the stack is inspected (without modification) in search for a catch handler. Each stack frame is examined by the relevant personality function.[12] If an appropriate catch handler is found in this phase, unwinding continues with a second phase; otherwise, an unwinder error is reported back to the throwing function. Unwinder errors usually cause program termination. In the second phase, the stack is examined again, and a personality function is invoked again for each frame. In this phase, cleanup handlers come into play. If any handler is

[7]Short for Virtual File System, since in a verification environment, the system under test must not access the real filesystem or any other part of the outside environment.

[8]The execution stack of a (C++) program consists of stack frames, each holding context of a single entry into some function. It contains local variables, a return address and register values which need to be restored upon return.

[9]*Application Binary Interface*, a low-level interface between program components on a given platform.

[10]There are many implementations of the C++ runtime library, which, besides exception support code, provides additional features such as RTTI. Each implementation is usually tied to a particular C++ standard library. Commonly used implementations on Unix-like systems are libsupc++, which comes with libstdc++ and the GCC compiler, and libc++abi, which is tied to libc++ used by some builds of clang and by DIVINE.

[11]In fact, the personality function can also decide to skip cleanup handlers, but this is not common.

[12]Different personality functions can be called for different frames, for example if the program consists of code written in different languages with exception support.

found (cleanup or catch), this fact is indicated to the unwinder, which performs the actual unwinding to the flagged frame. Once the control is transferred to the handler, it can either perform cleanup and resume propagation of the exception, or, if it is a catch handler, end the propagation of the exception. If exception propagation is resumed, the unwinder continues performing phase 2 from the point of the last executed handler. This is facilitated by storing the state of the unwinder within the exception object.

B. Unwind Tables

As mentioned in Section II-A, both the unwinder library and the language runtime depend on unwind tables for their work. The unwinder uses these tables to get information about stack layout in order to be able to unwind frames from it, and for detection which personality function corresponds to a frame. The personality function then uses the language-specific data area (LSDA) of these tables in its decision process.

While the unwinder part of the tables is unwinder- and platform-specific (it depends on stack layout), the LSDA is platform- and language-specific. For these reasons, unwind tables are not present in the LLVM IR; instead, they are generated by the appropriate code generator for any given platform, based on information in the `landingpad` instructions, and the personality attribute of functions. On Unix-like systems, the unwind tables are in the DWARF[13] format.

III. Execution of LLVM programs

In this section, we will look at how LLVM bitcode is executed by a model checker and how this execution is affected by addition of exception support. Unlike previous approaches, the technique described in this paper does not require any exception-specific intrinsic functions or hypercalls to be supported by the verifier. The exception-specific LLVM instructions can be implemented in the simplest possible way: `invoke` becomes equivalent to a `call` instruction followed by an unconditional branch. The `landingpad` instruction can be simply ignored by the verifier and `resume` instructions and calls of the `llvm.eh.typeid.for` intrinsic are both removed by the transformation described in Section IV. Moreover, the metadata required by `libc++abi` are likewise generated by the LLVM transformation and this process is completely transparent to the verifier.

In addition to support for LLVM, the unwinder (described in more detail in Section V) requires the ability to traverse and manipulate the stack and read and write LLVM registers associated with a given stack frame. Finally, it needs access to a representation of the bitcode for a given function. All those abilities are part of the DiVM specification [14] and are generally useful, regardless of their role in exception support.

The DiVM implementation in DIVINE 4 handles execution of LLVM instructions, LLVM intrinsic functions and DiVM-specific *hypercalls*.[14] Hypercalls exist to allocate memory, perform nondeterministic choice or to set DiVM's *control registers* (which contain, among other, the pointer to the currently executing stack frame). Additionally, DiVM performs safety checks, such as memory bound checking, and detects use of uninitialised values. However, DiVM hypercalls are intentionally low-level and simple and do not provide any high-level functionality, such as threading or standard C library functionality. Instead, those are provided by the DIVINE Operating System (DiOS) and the regular C and C++ standard libraries.

The most important purpose of DiOS is to provide threading support. To this end, DiOS provides a *scheduler*, which is responsible for keeping track of threads and their stacks and for (nondeterministically) deciding which thread should execute next. This scheduler is invoked repeatedly by the verifier to construct the state space. The scheduler fully determines the behaviour (or even presence) of concurrency in the verified program: while DiOS provides asynchronous, preemptive parallelism typical of modern operating systems, it is also possible to implement cooperative or synchronous schedulers instead.

A. Stack Layout and Control Registers

A DiVM program can have multiple stacks, but only one of them can be active at any given time (a pointer to the active stack is kept in a DiVM control register). The active stack is normally either the kernel stack or the stack that belongs to the active thread which was selected by the scheduler. Switching of stacks (and program counters) is performed by the `control` hypercall which manipulates DiVM control registers.

Traditionally, stack is represented as a continuous block of memory which contains an activation frame for each function call. In DiVM, the stack is not continuous; instead, it is a singly-linked list of activation frames, each of which points to its caller. This has multiple advantages: first, it is easy to create a stack frame for a function, for example when DiOS needs to create a new thread; additionally, the linked-list-organized stack is a natural match for the graph representation of memory which DiVM mandates, and therefore can be saved more efficiently [14]. Additionally, this way the stack may be nonlinear, and the unwinder can use this feature to safely transfer control to a cleanup block while the unwinder frame is still on the stack. Later, the handler can return control to the unwinder frame and the unwinder can continue its execution. This would be impossible with a continuous stack since cleanup code is allowed to call arbitrary functions and frames of those functions would overwrite the frame of the unwinder. For this reason, on traditional platforms, the unwinder needs to store its entire state in the exception object, while in DiVM,

[13]DWARF is a standard for debugging information designed for use with ELF executables. It is used on most modern Unix-like systems.

[14]Intrinsic functions are provided by LLVM as a light-weight alternative to new instructions. Such functions are recognized and translated by LLVM itself, as opposed to "normal" functions that come from libraries or the program. Likewise, DiVM provides hypercalls, which are functions that are, in addition to LLVM intrinsics, recognized by DiVM.

it can simply retain its own activation frame. An illustration of how the stack looks while the unwinder is active is shown in Figure 3.

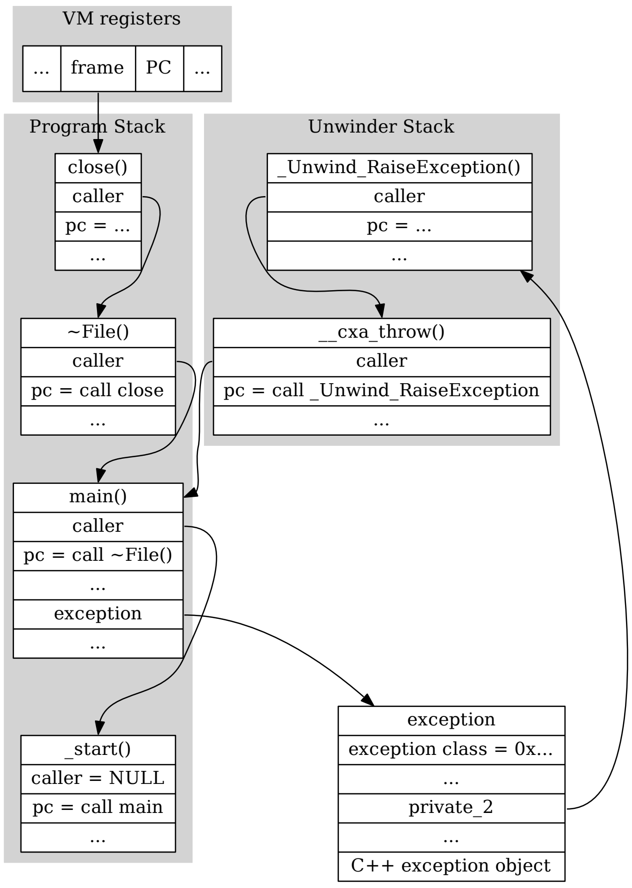


Fig. 3. In this figure we can see a stack of a program which is running cleanup block in the `main` function. The cleanup block calls the destructor of `File` structure, which in turn calls the `close` function (which is the current active function). Furthermore, the cleanup handler can access the exception object which contains a pointer to the stack of the unwinder. This pointer is used by the implementation of the `resume` instruction to jump back to the unwinder and continue phase 2 of the unwinding.

IV. The LLVM Transformation

The C++ runtime library (`libc++abi` in our case), needs access to the LSDA section of unwind tables (a pointer to this metadata section is accessible through the unwinder interface). This section contains DWARF-encoded exception tables, which are normally generated together with the executable by the compiler backend (code generator). Unfortunately, the generator of DWARF exception tables in LLVM is closely tied to the machine code generator and cannot be used to generate DWARF-formatted exception tables for verification purposes. For this reason, we have implemented a small LLVM transformation which processes the information in `landingpad` instructions and generates LLVM constants which contain the DWARF-formatted LSDA data. A reference to one such constant is attached to each function in the bitcode file.

To improve efficiency, LLVM does not directly use RTTI type info pointers within the landing blocks to decide which exception handlers should run. RTTI objects are special C++ objects which are used to identify types at runtime and are emitted by the C++ frontend as constants. Due to the complexities of C++ type system, matching RTTI types against each other is expensive: a search in a pair of directed acyclic graphs is required. Moreover, since the RTTI matching must be already done in the personality function to decide which frames to unwind, the personality can also pre-compute a numerical index for the landing pad. This index, also called a *selector value* is then used as a shortcut to run an appropriate `catch` clause within the landing block, instead of re-doing the expensive RTTI matching. Since the `catch` handler is typically expressed in terms of typeinfo pointers, it needs to efficiently obtain the selector value from a type info pointer. For this purpose, LLVM provides a `llvm.eh.typeid.for` intrinsic, which obtains (preferably at compile time) the selector value corresponding to a particular type info pointer.

Therefore, besides generating the LSDA data, the transformation statically computes the values which correspond to `llvm.eh.typeid.for` calls and substitutes them into the bitcode. Since the purpose of `llvm.eh.typeid.for` is to translate from RTTI pointers to selector values, it is only required that the integer selector value chosen for a particular RTTI object is in agreement with the personality function. In our implementation, this is ensured by computing the selector values statically for both the LSDA (which is where personality function obtains them) and for `llvm.eh.typeid.for` at the same time.

Finally, the transformation rewrites all uses of the `resume` instruction to ordinary calls to `Resume`, a function which is part of `libunwind` (see also Table I).

V. The Unwinder

The unwinder in DIVINE is designed around the interface described in the Itanium C++ ABI documentation,[15] adopted by multiple vendors and across multiple architectures. The implementation is part of the runtime libraries shipped with DIVINE.[16] The unwinder builds upon a lower-level stack access API which is provided by DiOS under `sys/stack.h`.

Due to the stack layout used in DiVM (a linked list of frames, see also Section III-A), our unwinder is much simpler than usual. The main task of unwinding is handled by the `RaiseException` function, which is called by the language runtime when an exception is thrown. This function performs the two phase handler lookup described in Section II-A and it adheres to the Itanium ABI specification, with the following exceptions:

i. it checks that an exception is not propagated out of a function which has the nounwind attribute set, and reports verification error if this is the case;
ii. if the exception is a C++ exception and there is no handler for this exception type, the unwinder chooses

[15]https://mentorembedded.github.io/cxx-abi/abi.html
[16]`runtime/libc/functions/unwind.cpp`

Function	Description
`SetGR`	Store a value into a general-purpose register
`GetGR`	Read a value from a general-purpose register
`SetIP`	Stora a value into the program counter
`GetIP`	Read the value of the program counter
`RaiseException`	Unwind the stack
`Resume`	Continue unwinding the stack after a cleanup
`DeleteException`	Delete an exception object
`GetLSDA`	Obtain a pointer to the LSDA
`GetRegionStart`	Obtain a base for relative code pointers

TABLE I
A LIST OF C FUNCTIONS PROVIDED BY `LIBUNWIND`. IN C, ALL THE FUNCTIONS ARE PREFIXED WITH `_UNWIND_` TO PREVENT NAME CONFLICTS WITH USER CODE AND OTHER LIBRARIES (I.E. THE C NAME OF `SETGR` IS `_UNWIND_SETGR`).

nondeterministically whether it should or should not unwind the stack and invoke cleanup handlers.

The purpose of the first deviation is to check consistency of exception annotations (arising, for example, from a `nothrow` function attribute as available in GCC and in clang). The second modification allows DIVINE to check both allowed behaviours of uncaught exceptions in C++: the C++ standard specifies that it is implementation-defined whether the stack is unwound (and destructors invoked) when an exception is not caught.[17] Since the program may contain errors which manifest only under one of these behaviours, it is useful to be able to test both of them.

A. Low-Level Unwinding

The primary function of the unwinder described above is to find exception handlers; for the actual unwinding of frames, it uses a lower-level interface provided by DiOS. This interface consists of two functions: `__dios_jump`, which performs a non-local jump, possibly affecting both the program counter and the active frame, and `__dios_unwind`, which removes stack frames from a given stack. `__dios_unwind` is designed in such a way that it can unwind any stack, not only the one it is running on, and is not limited to the topmost frames (effectively, it removes frames from the stack's singly-linked list, freeing all the memory allocated for local variables that belong to the unlinked frames, along with the frames themselves[18]). The unwinder identifies values as local variables by looking at the instructions of the active function – the results of `alloca` instructions are exactly the addresses of local variables.

B. Unwinder Registers

When an exception is propagating, a personality function has to be able to communicate with the code which handles the exception. In C++, the communicated information includes the address of the exception object and a selector value which is later used by the handler. On most platforms, these values are passed to the handler using registers, which are manipulated using unwinder's `SetGR` function. This function can either set the register directly (if it is guaranteed not to be overwritten before the control is transferred to the handler), or save the value in a platform-specific way and make sure it is restored before the handler is invoked.

In LLVM (and hence in DiVM), there is no suitable counterpart to the general purpose registers of a CPU; instead, the values set by the personality function should be made available to the program in the return value of the `landingpad` instruction. This, however, requires the knowledge of the expected semantics of these registers. Currently, all users of the unwinder are expected to use the same registers as the C++ frontend in clang. That is, register 0 corresponds to the exception object and register 1 corresponds to a type index. This also directly maps to the return type of `landingpad` instructions and therefore the register values can be saved directly into the LLVM register corresponding to the particular `landingpad` that is about to be executed.

Registers other than 0 and 1 are currently not supported. In LLVM, in line with the above observation about clang and C++, there is a convention that `SetGR` indices correspond to indices into the result tuple of a `landingpad` instruction. As long as this convention is preserved by a particular language frontend and its corresponding runtime library (personality function), it is very easy to extend our unwinder to support this language. Finally, if a language frontend were instead to emit calls to `GetGR` in the handler, registers of this type can be stored in the unwinder `Context` directly.

C. Atomicity of the Unwinder

The unwinder performs rather complex operations and therefore throwing an exception can create many states, even when τ reduction [12] is enabled. However, many of these states are not interesting from the point of view of verification, as the operations performed by the unwinder are mostly thread-local and only the exception handlers (and possibly personality function) can perform globally visible actions. For this reason, the unwinder uses DiVM's atomic sections to hide most of its complexity.

Since an atomic section is implemented as an *interrupt mask* (i.e. a single flag indicating that an atomic section is executing) in DiVM, it is necessary to correctly maintain the state of this flag. In particular, it is required that the unwinder behaves reasonably even if it is called when the program is already in an atomic section. Consequently, care must be taken to restore the state of the atomic mask when the unwinder transfers control to a personality function or an exception handler. When the unwinder is first called, it enters an atomic section and saves the previous value of the interrupt mask. This will be the value the flag will be restored to when a personality function is first invoked. The mask is later re-acquired after the personality function returns and it is restored once more when the first handler is invoked. When the exception handler resumes (using the `resume` instruction), the atomic section is re-entered and its state saved so its state before the resume can

[17] Section 15.5, paragraph 9 of the C++ standard [5]

[18] When a function returns normally (due to a `ret` instruction), DiVM takes care of freeing the frame and its local variables (`alloca` memory).

be restored again for the next call to a personality function. This way, it is possible to safely throw an exception out of an atomic section, provided that the atomic section is exception-safe (that is, it has an exception handler which ends the atomic section if an exception is propagated out of it).

D. `longjmp` Support

Using the low-level unwinder interface described in Section V-A, it is easy to implement other mechanisms for non-local transfer of control. The functions `longjmp` and `setjmp`, specified as part of C89, are one such example.[19] The `setjmp` function can be used to save part of the state of the program, so that a later call to `longjmp` can restore the stack to the state it was in when `setjmp` was called. This way, `longjmp` can be used to remove multiple frames from the stack. When `longjmp` is called, the program behaves as if `setjmp` returned again, only this time it returns a different value (provided as an argument to `longjmp`).

The DIVINE implementation of `setjmp` saves the program counter and the frame pointer of the caller of `setjmp`. The `longjmp` function then uses this saved state, along with access to the text of the program, to set the return value of the `call` instruction corresponding to the `setjmp`. Afterwards, it unwinds the stack using the low-level stack access API from `sys/stack.h` and transfers control to the instruction right after the call to `setjmp`.

VI. Related Work

Primarily, we have looked at existing tools which support verification of C++ programs. Existence of an implementation is, to a certain degree, an indication that a given approach is viable in practice. We have, however, also looked at approaches proposed in the literature which have no implementations (or only a prototype) available.

A number of verification tools are based on LLVM and therefore have some support for C++. LLBMC [15] and NBIS [6] are LLVM-based bounded model checkers which target mainly C and have no support for exceptions or the C++ standard library. VVT [7] is a successor of NBIS which uses either IC3 or bounded model checking and has limited C++ support, but it does not support exceptions. Furthermore, KLEE [3] and KLOVER [9] are LLVM-based tools for test generation and symbolic execution. KLOVER targets C++ and according to [9] has exception support, but it is not publicly available. On the other hand, KLEE focuses primarily on C and its C++ support is rather limited and it has no exception support.

Both CBMC [4, 8] and ESBMC [11] bounded model checkers support C++ (but neither appears to support the standard library) and they include support for exceptions. However, in CBMC, the support for exceptions is limited to throwing and catching fundamental types.[20] In our survey of tools for verification of C++ programs, ESBMC has by far the best exception support: the latest version can deal with most, but not all[21], types of exception handlers and even with exception specifications. Finally, DIVINE 3 [13] also comes close to full support for exceptions, but lacks support for exception specifications. Overall, this survey suggests that all current implementations of C++ exceptions in verification tools are incomplete and confirms that using an existing, standards-compliant implementation in a verification tool is indeed quite desirable.

Finally, it is also possible to transform a C++ program with exceptions into an equivalent program which only uses more traditional control flow constructs. This approach was taken in [10], with the goal of re-using existing analysis tools without exception support. While this approach is applicable to a wide array of verification tools, it is also incompatible with re-use of existing exception-related runtime library code. As such, it offers a very different set of tradeoffs than our current approach. Moreover, the translation cost is far from negligible, and also affects code that does not directly deal with exceptions (i.e. it violates the zero-cost principle of modern exception handling). Unfortunately, we were unable to evaluate this approach, since there are no publicly available tools which would implement it.

VII. Evaluation

In order to asses the viability of our approach, we have executed a set of benchmarks in various configurations of DIVINE 4. The benchmarks were executed on quad-core Xeon 5130 clocked at 2 GHz and with 16GB of RAM. We have measured the wall time, making all 4 cores available to the verifier.

A. Benchmark Models

The set of models we have used for this comparison consists of 831 model instances, out of which we picked the 794 that do not contain errors. The reason for this is that the execution time is much more variable when a given program contains an error, since the model checking algorithm works on the fly, stopping as soon as the error is discovered.

Majority of the valid models (777) are C++ programs of varying complexity, while the 17 models in the svc-pthread category are concurrent programs written in plain C with pthreads. Since our implementation of the pthread API is done in C++, the impact of exception support on verification of C programs is also relevant. The “alg” category includes sequential algorithmic and data structure benchmarks, the “pv264” category contains unit tests for student assignments in a C++ course, the “iv112” category contains unit tests for concurrent data structures and other parallel programs (again assignment problems in a C++ course), “libcxx” contains a selection of the

[19]Implemented in `runtime/libc/includes/setjmp.h` and `runtime/libc/functions/setjmp/`.

[20]A simple test which throws and tries to catch an exception object crashes CBCM 5.6.

[21]ESBMC 3.0 is unable to determine that an exception ought to be caught when the `catch` clause specifies a type which is a virtual base class in a diamond-shaped hierarchy and the object thrown is of the most-derived type of the diamond. This suggests that ESBMC uses its own implementation of RTTI support code, which is somewhat incomplete, compared to production implementations.

category	#mod	time (D4)	time (D3)	states (D4)	states (D3)
alg	9	3:52	3:51	543.3 k	543.3 k
pv264	13	1:34	1:32	183.0 k	183.0 k
iv112	11	25:58	25:57	3743 k	3743 k
libcxx	425	42:15	42:09	2182 k	2182 k
bricks	292	3:04:25	2:56:55	6271 k	6251 k
divine	3	6:20	6:18	1040 k	1040 k
cryptopals	3	0:01	0:01	1943	1943
llvm	12	36:36	36:27	3865 k	3865 k
svc-pthread	17	16:47	16:41	1685 k	1685 k
total	794	5:21:44	5:13:49	20.1 M	20.0 M

TABLE II
COMPARISON OF THE NEW EXCEPTION CODE WITH A DIVINE-3-STYLE VERSION.

category	#mod	time (D4)	time (stub)	states (D4)
alg	9	3:52	3:52	543.3 k
pv264	13	1:34	1:34	183.0 k
iv112	11	25:58	26:00	3743 k
libcxx	392	41:56	41:54	2179 k
bricks	192	35:30	35:21	2378 k
divine	3	6:20	6:19	1040 k
cryptopals	3	0:01	0:01	1943
llvm	12	36:36	36:28	3865 k
svc-pthread	17	16:47	16:43	1685 k
total	661	2:52:30	2:52:08	16.2 M

TABLE III
COMPARISON OF THE NEW EXCEPTION CODE AGAINST STUBBED EXCEPTIONS. COMPARED TO TABLE II, IN THIS CASE 133 MODELS FAILED DUE TO THE STUBS. STATE COUNTS ARE IDENTICAL FOR ALL MODELS.

`libc++` testsuite (with focus on exception support coverage), “bricks” contains unit tests for various C++ helper classes, including concurrent data structures, “divine” contains unit tests for a concurrent hashset implementation used in DIVINE, “cryptopals” contains solutions of the cryptopals problem set[22], the “llvm” category contains programs from the LLVM test-suite[23] and finally, the “svc-pthread” category includes pthread-based C programs from the SV-COMP benchmark set. In most of the programs, it was assumed that `malloc` and `new` never fail, with the notable exception of part of the “bricks” category unit tests. The tests where `new` failures are allowed are especially suitable for evaluating exception code, in particular where multiple concurrent threads are running at the time of the possible failure.

B. Comparison to Builtin Exception Support

In addition to the approach presented in this paper, we have implemented the approach described in [13] in the context of DIVINE 4. This allowed us to directly measure the penalty associated with the present approach, which is more thorough and less labour-intensive at the same time. Our expectation was that this would translate to slower verification, since the off-the-shelf code is more complex than the corresponding hand-tailored version used in [13]. In line with this expectation, we set the criterion of viability: we would consider a slowdown of at most 10 % to be an acceptable price for the improved verification fidelity, and convenience of implementation. Since other resource consumption (especially memory) of verification is typically proportional to state space size, we have used the number of states explored as an additional metric. The expected effect on the shape (and, by extension, size) of the state space should be smaller than the effect on computation time (most of the additional complexity is related to computing a single transition). We believe that an acceptable penalty in this metric would be about 2 % increase.

As can be seen in Table II, the time penalty on our chosen model set is very acceptable – just shy of 2.6 % – and the state space size is within 1 % of the older approach [13]. We believe that this small penalty is well justified by the superior verification properties of the new approach.

C. Comparison to Stub Exceptions

The second alternative approach is to consider any thrown exception an error, regardless of whether it is caught or not. This can be achieved much more easily than real support for exceptions, since we can simply replace the entire `libunwind` interface with stubs which raise an error and refuse to continue. This approach only works for models which do not actually throw any exceptions during their execution. The results of this comparison are shown in Table III – the verification time is nearly identical and the state spaces are entirely so. This is in line with expectations: in those models, catch blocks are present but never executed. Since the proposed approach does not incur any overhead until an exception is actually thrown, we would not expect a substantial time difference.

D. Comparison to No Exceptions

Finally, the last alternative is to disable exception support in the C++ frontend entirely. In `clang`, this is achieved by compiling the source code with the `-fno-exceptions` flag. In this case, the LLVM bitcode contains no exception-related artefacts at all, but many programs fail to build. Additionally, a number of programs in the “bricks” category contain exception handlers for memory allocation errors[24] and therefore exit cleanly upon memory exhaustion. Even though some of those programs can be compiled with `-fno-exceptions`, they now contain an error (a null pointer dereference) which is not present when they are compiled the standard way. Those programs were therefore excluded from the comparison. The summary of this comparison can be found in Table IV – the time saved for models where `-fno-exceptions` is applicable is again quite small, less than 13 %. In this case, the

[22]http://cryptopals.com
[23]http://llvm.org/svn/llvm-project/test-suite/trunk/SingleSource/Benchmarks/Shootout

[24]In this case, the handler is installed using `std::set_terminate`, which is available even when `-fno-exceptions` is given. The situation would be similar if only parts of the program were compiled with `-fno-execptions`. In particular, the problem is that the standard library, if compiled with `-fno-exceptions`, cannot throw, and must therefore behave differently in those scenarios, affecting the behaviour of the user program.

category	#mod	time (D4)	time (nxc)	states
alg	1	0:24	0:23	34.2 k
pv264	1	0:00	0:00	57
iv112	10	23:58	22:06	3571 k
libcxx	393	41:57	40:44	2180 k
svc-pthread	17	16:47	15:42	1685 k
total	423	1:23:33	1:19:21	7504 k

TABLE IV
COMPARISON OF THE NEW EXCEPTION SUPPORT AGAINST A CASE WHERE `-fno-exceptions` WAS USED TO COMPILE THE SOURCES AND LIBRARIES. IN THIS CASE, IT WAS ONLY POSSIBLE TO VERIFY 423 MODELS FROM THE SET (I.E. 371 MODELS ARE MISSING FROM THE COMPARISON). STATE COUNTS ARE IDENTICAL FOR ALL MODELS.

difference is due to the changes in control flow of the resulting LLVM bitcode. Since `call` is not a terminator instruction (unlike `invoke`), the *local* control flow in a function is negatively affected by the presence of `invoke` instructions: more branching is required, and this slows down the evaluator in DiVM. While it is easy to see if a given program can be compiled with `-fno-exceptions`, it is typically much harder to ensure that its behaviour will be unchanged. For this reason, we do not consider the time penalty in verification of this type of programs a problem.

E. Re-usability

As outlined in Section I-F, the two components directly involved in exception support are comparatively small and well isolated. The LLVM transformation is fully re-usable with any LLVM-based tool. The unwinder, on the other hand, relies on the capabilities of DiVM. However, there is no need for hypercalls specific to exception handling and therefore, the implementation work is essentially transparent to DiVM. The capabilities of DiVM required by the unwinder are limited to the following: linked-list stack representation, runtime access to the program bitcode and 2 hypercalls: `__vm_control` and `__vm_obj_free`. More details about DiVM can be found in [14].

Finally, adding support for a new type of exceptions is also much simpler in this approach – no modifications to DiVM (or any other host tool) are required: only the two components described in this paper may need to be modified.

VIII. CONCLUSION

In this paper, we have discussed an approach to extending an LLVM-based model checker with C++ exception support. We have found that re-using an existing implementation of the runtime support library is a viable approach to obtain complete, standards-compliant exception support. A precondition of this approach is that the verification tool is flexible enough to make stack unwinding possible. The DiVM language, on which the DIVINE model checker is based, has proven to be a good match for this approach, due to its simple and explicit stack representation, along with a suitable set of control flow primitives.

We also performed a survey of tools based on partial or complete reimplementations of C++ exception support routines and found that in each tool, at least one edge case is not well supported. In contrast to this finding, with our approach, all those edge cases are covered "for free", that is, by the virtue of re-using an existing, complete implementation. Contrary to the prediction made in [13], we have found that with a suitable target language, implementing a new unwinder can be relatively simple. The unwinder implementation described in this paper is only about 350 lines of C++ code, while it would be impossible to implement without verifier modifications in DIVINE 3. Therefore, we can conclude that with the advent of the DiVM specification [14] and its implementation in DIVINE 4, re-implementing the `libunwind` API and re-using `libc++abi` became a viable strategy to provide exception support.

REFERENCES

[1] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. DiVinE 3.0 – an explicit-state model checker for multithreaded C & C++ programs. In *CAV*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.

[2] Dirk Beyer. Reliable and Reproducible Competition Results with BenchExec and Witnesses Report on SV-COMP 2016. In *TACAS*, pages 887–904. Springer, 2016. ISBN 978-3-662-49673-2. doi: 10.1007/978-3-662-49674-9_55.

[3] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, (OSDI 2008)*, pages 209–224. USENIX Association, 2008.

[4] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, pages 168–176. Springer, 2004. ISBN 978-3-540-24730-2. doi: 10.1007/978-3-540-24730-2_15.

[5] ISO C++ Standards Committee. Standard for Programming Language C++. Working Draft N4296. Technical report, ISO IEC JTC1/SC22/WG21, 2014.

[6] Henning Günther and Georg Weissenbacher. Incremental Bounded Software Model Checking. In *SPIN*. ACM, 2014.

[7] Henning Günther, Alfons Laarman, and Georg Weissenbacher. Vienna verification tool: IC3 for parallel software - (competition contribution). In *TACAS*, pages 954–957, 2016. doi: 10.1007/978-3-662-49674-9_69.

[8] Daniel Kroening and Michael Tautschnig. CBMC – C bounded model checker. In *TACAS*, pages 389–391. Springer, 2014. ISBN 978-3-642-54862-8. doi: 10.1007/978-3-642-54862-8_26.

[9] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In *CAV*, volume

6806 of *LNCS*, pages 609–615. Springer, 2011. ISBN 978-3-642-22109-5.

[10] Prakash Prabhu, Naoto Maeda, Gogul Balakrishnan, Franjo Ivančić, and Aarti Gupta. Interprocedural exception analysis for C++. In *ECOOP*, volume 6813 of *LNCS*, pages 583–608. Springer, 2011. ISBN 978-3-642-22654-0.

[11] Mikhail Ramalho, Mauro Freitas, Felipe Sousa, Hendrio Marques, Lucas Cordeiro, and Bernd Fischer. SMT-Based Bounded Model Checking of C++ Programs. In *ECBS*, pages 147–156. IEEE Computer Society, 2013. ISBN 978-0-7695-4991-0.

[12] Petr Ročkai, Jiří Barnat, and Luboš Brim. Improved state space reductions for LTL model checking of C & C++ programs. In *NASA Formal Methods*, volume 7871 of *LNCS*, pages 1–15. Springer, 2013.

[13] Petr Ročkai, Jiří Barnat, and Luboš Brim. Model checking C++ programs with exceptions. *Science of Computer Programming*, 128:68 – 85, 2016.

[14] Petr Ročkai, Vladimír Štill, Ivana Černá, and Jiří Barnat. DiVM: Model checking with LLVM and graph memory. 2017. URL https://arxiv.org/abs/1703.05341. Preliminary version.

[15] Carsten Sinz, Florian Merz, and Stephan Falke. LLBMC: A bounded model checker for LLVM's intermediate representation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *LNCS*, pages 542–544. Springer, 2012. ISBN 978-3-642-28755-8. doi: 10.1007/978-3-642-28756-5_44.

Model Checking of C and C++ with DIVINE 4[⋆]

Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera,
Henrich Lauko, Jan Mrázek, Petr Ročkai, Vladimír Štill

Faculty of Informatics, Masaryk University
Brno, Czech Republic
divine@fi.muni.cz

Abstract. The fourth version of the DIVINE model checker provides a modular platform for verification of real-world programs. It is built around an efficient interpreter of LLVM code which, together with a small, verification-oriented operating system and a set of runtime libraries, enables verification of code written in C and C++.

1 Introduction

Building correct software is undoubtedly an important goal for software developers and we firmly believe that formal verification methods can help in this endeavour. In particular, explicit-state model checking promises to put forth a deterministic testing procedure for non-deterministic problems (such as parallel programs or tests which use fault injection). Moreover, it is quite easy to integrate into common test-based workflows. The latest version of DIVINE aims to make good on these promises by providing an efficient and versatile tool for analysis of real-world C and C++ programs.

2 DIVINE 4 Architecture

The most prominent feature of DIVINE 4 is that the runtime environment for the verified program (i.e. support for threads, memory allocation, standard libraries) is not part of the verifier itself: instead, it is split into several components, separated by well-defined interfaces (see Figure 1). The three most important components are: the DIVINE Virtual Machine (DiVM), which is an interpreter of LLVM code and provides basic functionality such as non-determinism and memory management; the DIVINE Operating System (DiOS), which takes care of thread management and scheduling; and finally libraries, which implement standard C, C++ and POSIX APIs. The libraries use syscalls to communicate with DiOS and hypercalls to communicate with DiVM.

The verification core below DiVM is responsible for verification of safety and liveness properties and uses DiVM to generate the state space of the (non-deterministic) program.

[⋆] This work has been partially supported by the Czech Science Foundation grant No. 15-08772S and by Red Hat, Inc.
[†] The final publication will be available at Springer.

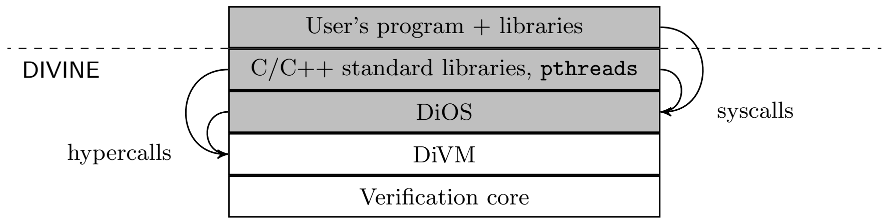

User's program + libraries
DIVINE
C/C++ standard libraries, pthreads
DiOS
syscalls
hypercalls
DiVM
Verification core

Fig. 1. Overview of the architecture of DIVINE 4. The shaded part consists of LLVM code which is interpreted by DiVM.

2.1 DIVINE Virtual Machine (DiVM)

The basic idea of DiVM is to provide the bare minimum required for efficient model checking of LLVM-based programs. To this end, it executes instructions, manages memory, implements non-deterministic choice, and (with the help of program instrumentation) keeps track of visible actions performed by the program. To the verification core, DiVM provides support for saving and loading snapshots of the program state and for generating successors of a given program state.

When the user program is executed in DiVM, it will be typically supported by a runtime environment (which itself executes on top of DiVM). This environment is expected to supply a *scheduler*, a procedure invoked by DiVM to explore the successors of a given program state. The scheduler's primary responsibility is thread management. It can, for example, implement asynchronous thread interleaving by managing multiple stacks and non-deterministically choosing which to execute. This design allows DiVM to be small, minimising the space for errors in this crucial part of the verifier. Moreover, it allows for greater flexibility, since it is usually much easier to program for DiVM then to change DiVM itself.

DiVM uses a graph to represent the memory of a program: nodes correspond to memory objects (e.g. results of allocation, global variables) and edges to pointers between these objects. Each program state corresponds to one such graph. When exploring the state space, those graphs are stored, hashed and compared directly (i.e. they are not converted to byte vectors). This graph representation allows DiVM to handle programs with dynamic heap allocation efficiently.

Out of the box, memory access in DiVM is subject to sequentially consistent semantics. Nevertheless, analysis under relaxed memory semantics can be added to DIVINE by the means of program transformations on the level of LLVM, as outlined in [7].

More details about DiVM, including an experimental evaluation, can be found in [6].

2.2 DIVINE Operating System (DiOS)

DiOS supplies both a scheduler, which is invoked by DiVM, and a POSIX-like environment for the libraries and the user program. To this end, DiOS exposes a syscall interface to `libc`, in a manner similar to common operating systems. Currently, DiOS implements asynchronous parallelism with threads and supports syscalls which cover an important subset of the POSIX file system interface (provided by the integrated virtual file system). Additional syscalls make thread management and DiOS configuration possible.

2.3 State Space Reductions

To be able to verify nontrivial C or C++ programs, DIVINE 4 employs heap symmetry reduction and τ reduction [5]. The latter reduction targets parallel programs and is based on the observation that not all actions performed by a given thread are visible to other threads. These local, invisible actions can be grouped and executed atomically. In DIVINE, actions are considered visible if they access shared memory. As DiVM has no notion of threads, the *shared status* of a memory object is partially maintained by DiOS. However, DiVM transparently handles the propagation of shared status to objects reachable from other shared objects.

It is desirable that threads are only switched at well-defined points in the instruction stream: in particular, this makes counterexamples easier to process. For this reason, DIVINE instruments the program with interrupt points prior to verification. DiVM will then only invoke the scheduler at these explicit interrupt points, and only if the program executed a shared memory access since the previous interrupt. To ensure soundness, these interrupt points are inserted such that there cannot be two accesses to shared memory without an interrupt point between them.

To further reduce the size of the state space of parallel programs, DIVINE performs heap symmetry reduction. That is, heaps that differ only in concrete values of memory addresses are considered identical for the purpose of verification. On top of that, DIVINE 4 also employs static reductions which modify the LLVM IR. However, it only uses simple transformations which are safe for parallel programs and which cause minimal overhead in DiVM.

2.4 C and C++ Language Support

For practical verification of C and C++ code, it is vital that the verifier has strong support for all language features and for the standard libraries of these languages, allowing the user to verify unmodified code. DIVINE achieves this by integrating ported implementations of existing C and C++ standard libraries. Additionally, an implementation of the POSIX threading API was developed specifically for DIVINE. These libraries together provide full support for C99 and C++14 and their respective standard libraries.

As DiVM executes LLVM instructions and not C or C++ directly, the program needs to be translated to LLVM IR and linked with the aforementioned libraries. This is done by an integrated compiler, based on the clang C/C++ frontend library. How a program is processed by DIVINE is illustrated in Figure 2. The inputs to the build are a C or a C++ program and, optionally, a

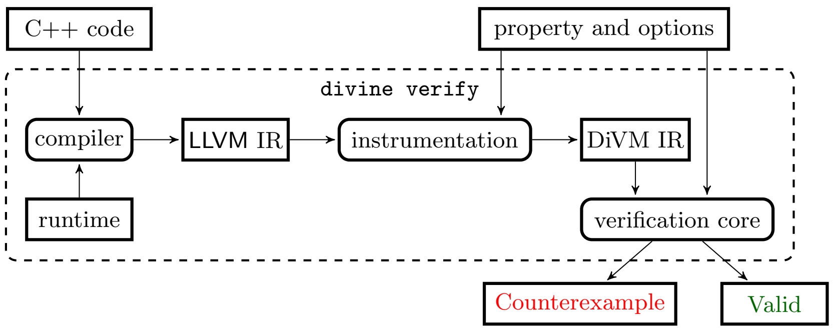


Fig. 2. Verification workflow of the `divine verify` command when it is given a C++ file as an input. Boxes with rounded corners represent stages of input processing.

specification of the property to be verified. The program is first compiled and linked with runtime libraries, producing an LLVM IR module. This module is then instrumented to facilitate τ reduction (see Section 2.3) and annotated with metadata required for exception support [8]. The instrumented IR is then passed to the verification algorithm, which uses DiVM to evaluate it. Finally, the verification core (if provided with sufficient resources) either finds an error and produces a counterexample, or concludes that the program is correct.

2.5 Property Specification

DIVINE 4 supports a range of safety properties: detection of assertion violations, arithmetic errors, memory errors (e.g. access to freed or otherwise invalid memory), use of uninitialised values in branching, and `pthreads` locking errors.

The libraries shipped with DIVINE can simulate memory allocation failures and spurious wake-ups on `pthreads` conditional variables. The allocation failure simulation can be disabled on DIVINE's command line.

Monitors and Liveness More complex properties can be specified in the form of *monitors*, which are executed synchronously by DiOS every time a visible action (as determined by τ reduction) occurs. This allows such monitors to observe globally visible state changes in the program, and therefore to check global assertions or liveness properties (using a Büchi accepting condition). Moreover, it is also possible to disallow some runs of the program, i.e. the monitor can specify that the current run of the program should be abandoned and ignored.

In order to check LTL properties in DIVINE, the LTL formula has to be translated to a Büchi automaton encoded as a monitor in C++. This translation can be done automatically by an external tool, `dipot`,[1] which internally uses SPOT [1] to process the LTL formula.

[1] available from `https://github.com/xlauko/dipot`

2.6 Interactive Program Simulator

Model checkers traditionally provide the user with a counterexample: a trace from the initial state to the error state. However, with real-world programs, the presentation of this trace is critical: the user needs to be able to understand the complex data structures which are part of the state, and how they evolve along the error trace. To help with this task, DIVINE now contains an interactive simulator that can be used to perform the steps of the program which led to the error and to inspect values of program variables at any point in the execution [4].

2.7 Major Changes Compared to DIVINE 3

Compared to DIVINE 3, the new version comes with several improvements. From architectural point of view, the most important changes are the introduction of DiVM and DiOS and the graph-based representation of program memory [6]. From user perspective, the most important changes include better support for C++ and its libraries, an improved compilation process which makes it easier to compile C and C++ programs into LLVM IR, an interactive simulator of counterexamples, and support for simulation of POSIX-compatible file system operations.

3 Usage and Evaluation

DIVINE is freely available online[2], including source code, a user manual, and examples which demonstrate the most important features of DIVINE. In addition to the source code, it is also possible to download a pre-built binary for 64bit Linux (which also works on Windows Subsystem for Linux), or a virtual machine image with DIVINE installed (available in 2 formats, OVA for VirtualBox, and VDI for QEMU and other hypervisors). If you choose to build DIVINE from source code, please refer to the user manual[3] for details.

3.1 Using DIVINE

Consider the code from Figure 3 and assume it is saved in a file named `test.cpp`. Assuming that DIVINE is installed[4], the code can be verified by simply executing `divine verify test.cpp`. DIVINE will report an invalid write right after the end of the `x` array. You can observe that the output of `printf` is present in the `error trace` part of DIVINE's output. Moreover, toward the end, the output includes stack traces of all running threads. In this case, there are two threads, the main thread of the program and a kernel thread, in which the fault handler is being executed.

[2] `https://divine.fi.muni.cz/2017/divine4/`
[3] `https://divine.fi.muni.cz/manual.html`
[4] the binary has to be in a directory which is listed in the `PATH` environment variable

```
#include <cstdio>
#include <cassert>
void foo( int *array ) {
    for ( int i = 0; i <= 4; ++i ) {
        printf( "writing at %d\n", i );
        array[i] = 42;
    }
}
```

```
int main() {
    int x[4];
    foo( x );
    assert( x[3] == 42 );
}
```

Fig. 3. Example C++ code which creates an array `x` of size 4 (on the stack) and then, in function `foo`, writes into this array. `foo` does, however, attempt to write one element past the array, which would normally overwrite the next entry on the stack but not cause an immediate program failure. In DIVINE, this error is detected and reported.

If we wanted to inspect the error state in more detail, we could use DIVINE's simulator. First, we need a way to identify the error state: the counterexample contains a line which reads `choices made: 0^182`; the sequence of numbers after the colon is a sequence of non-deterministic choices made by DIVINE. We can now run `divine sim test.cpp` and execute `trace 0^182` (replacing the sequence of choices with the ones actually produced by `divine`). This makes the simulator stop after the last non-deterministic choice before the error. The error location can be inspected by executing `stepa`, which tells DIVINE to perform a single atomic step (unless an error occurs, in which case it stops as soon as the error is reported). It is now possible to examine the frame of the error handler, although it is more useful to move to the frame which caused the error by executing the `up` command. At this point, local variables can be inspected by using `show`.

Please consult the user manual for more detailed information on using DIVINE. Additionally, `divine help` and the `help` command in the simulator provide short descriptions of all available commands and switches.

3.2 Evaluation

We have evaluated DIVINE 4 on a set of more than 900 benchmarks from various sources, including parallel and sequential tests of parts of C and C++ standard libraries, the `pthread` library, part of the SV-COMP `pthread` benchmark set, and programs from various programming courses. We have compared DIVINE 4 to DIVINE 3 (an older version of DIVINE, also an explicit-state model checker) and ESBMC 4.1 [3] (an SMT-based symbolic model checker).

From our benchmark set, DIVINE 3 was able to process 457 benchmarks in 7 hours, while DIVINE 4 processed the same 457 benchmarks in 1 hour and 5 minutes. ESBMC was only able to process 60 benchmarks, mostly due to limitations of its C++ support and worse performance on threaded benchmarks. ESBMC took 2 hours 43 minutes, while DIVINE 4 only took 10 minutes on the same subset. In all cases, there was a timeout of 2 hours and benchmarks which timed out were not included in the results. More details are available online.[5]

[5] `https://divine.fi.muni.cz/2017/divine4/`

Overall, DIVINE 4 showed substantial improvement over DIVINE 3, both in terms of speed as well as C++ support. Compared to ESBMC, DIVINE 4 has again the advantage of better C++ support (partially due to usage of clang compiler whereas ESBMC has custom C++ frontend) and additionally better performance on programs with threads.

4 Conclusion and Future Work

In this paper, we have introduced DIVINE 4, a versatile explicit-state model checker for C and C++ programs, which can handle real-world code using an efficient LLVM interpreter which has strong support for state space reductions. The analysed programs can make use of the full C99 and C++14 standards, including the standard libraries.

In the future, we would like to take advantage of the new program representation and versatility of DiVM to extend DIVINE with support for programs with significant data non-determinism, taking advantage of abstract and/or symbolic data representation, building on ideas introduced in SymDIVINE [2]. We would also like to add support for verification of concurrent programs under relaxed memory models, based on [7].

References

1. Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Étienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Automated Technology for Verification and Analysis (ATVA 2016)*, volume 9938 of *LNCS*, pages 122–129. Springer International Publishing, October 2016.
2. Jan Mrázek, Petr Bauch, Henrich Lauko, and Jiří Barnat. SymDIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration. In *Model Checking Software (SPIN 2016)*, pages 208–213. Springer International Publishing, 2016.
3. Mikhail Ramalho, Mauro Freitas, Felipe Sousa, Hendrio Marques, Lucas Cordeiro, and Bernd Fischer. SMT-Based Bounded Model Checking of C++ Programs. In *Engineering of Computer Based Systems (ECBS)*, pages 147–156. IEEE Computer Society, 2013.
4. Petr Ročkai and Jiří Barnat. A Simulator for LLVM Bitcode. *Preliminary version, ArXiv: 1704.05551*, 2017.
5. Petr Ročkai, Jiří Barnat, and Luboš Brim. Improved State Space Reductions for LTL Model Checking of C & C++ Programs. In *NASA Formal Methods (NFM 2013)*, volume 7871 of *LNCS*, pages 1–15. Springer, 2013.
6. Petr Ročkai, Vladimír Štill, Ivana Černá, and Jiří Barnat. DiVM: Model Checking with LLVM and Graph Memory. *Preliminary version, ArXiv: 1703.05341*, 2017.
7. Vladimír Štill, Petr Ročkai, and Jiří Barnat. Weak Memory Models as LLVM-to-LLVM Transformations. In *Mathematical and Engineering Methods in Computer Science (MEMICS 2015)*, volume 9548 of *LNCS*, pages 144–155. Springer, 2016.
8. Vladimír Štill, Petr Ročkai, and Jiří Barnat. Using Off-the-Shelf Exception Support Components in C++ Verification. In *IEEE International Conference on Software Quality, Reliability and Security (QRS 2017) (to appear)*, 2017.