Faculty of Informatics, Masaryk University

LLVM Transformations for Model Checking

Master's thesis

Vladimír Štill

Brno, 2016

Declaration

Thereby I declare that this thesis is my original work, which I have created on my own. All sources and literature used in writing the thesis, as well as any quoted material, are properly cited, including full reference to its source.

Advisor: doc. RNDr. Jiří Barnat, Ph.D.

Abstract

This work focuses on application of LLVM transformations as a preprocessing step for verification of real-world C and C++ programs using the explicit-state model checker DIVINE [6]. We demonstrate that LLVM transformations can be used for extension of verifier capabilities and for reduction of the state space size.

In the case of extension of verifier capabitilies, the main focus is on verification under relaxed memory models, this is a continuation of the work started in [42]. We extend the previous transformation to enable verification of wider range of safety properties, support code with atomic instructions, support more relaxed memory models than total store order, and improve state space size of the programs which use this transformation. The final implementation of this transformation is evaluated and compared with the previous implementation.

For state space reductions, we propose the concept of optimizations which preserve verified property, evaluate some of these optimizations, and propose additional transformations which can be implemented as future work.

Keywords

Formal Verification, C, C++, LLVM, Model Checking, Parallel, DIVINE, LART, LLVM Transformation, Weak Memory Models, Total Store Order, Implementation.

Acknowledgements

I would like to thank all the people in the ParaDiSe laboratory for the support and for the pleasant work environment. Namely, I would like to thank Jiří Barnat for advising this thesis and Petr Ročkai for consulting with me. I would also like to thank all the people who contributed to DIVINE.

Finally, I would like to thank my friends and my family for supporting me and having patience with me.

Contents

Chapter 1

Introduction

With modern multi-core CPUs, multi-threaded programs are increasingly common and therefore the need to show their correctness, or at least find bugs in them is increasing. The problem is that testing of multi-threaded programs lacks a well established deterministic procedure. While common techniques, such as unit testing, can be applied to parallel programs, they are unable to reliably find bugs caused by data races.

The underlying reason is that data races occur when actions performed by threads in parallel are interleaved in an unexpected order which exposes the problem. This interleaving might be, however, quite rare and therefore it is often hard to reveal that particular erroneous interleaving during testing. Furthermore, even if the bug can be triggered by testing, it is often hard to reproduce and debug it, as common debugging approaches, including debuggers and logging often interfere with testing as they can hide the particular erroneous run.

1.1 Explicit-state Model Checking

Formal methods, explicit-state model checking using automata-base approach [17] in particular, can help in this regard. Explicit-state model checking allows us to explore all possible interleavings of parallel programs and therefore uncover even extremely rare data races. While pure explicit-state model checking requires programs to be closed (not have any inputs) it is still very helpful as it can be applied for example to unit tests and this combination yields deterministic testing procedure for parallel unit tests. Furthermore, there are techniques, such as control-explicit-data-symbolic model checking [3], which allow application of model checking to parallel programs with inputs.

On the other hand, formal methods come with a new set of issues, such as their computational complexity and their usability for software developer. For a verification tool to be useful to software developers, it is important to minimize any extra effort the developers have to put into usage of a such tool.

This effort was large with older generations of verifiers such as SPIN [18] and LTSmin [21], which required manual translation of the verified program into a modeling language such as ProMeLa. With the new generation of verifiers, such as DIVINE 3 [6], CBMC [22], and LLBMC [38], special-purpose languages for verification are no longer required. These tools support direct verification of widely used programming languages such as C and C++, either directly or using the LLVM intermediate representation [24] (an intermediate language which can be used in translation of many programming languages, including C and C++).

LLVM IR in particular is becoming input language of choice for many verification tools. This assembly-like language is simpler to work with than higher level languages, yet it maintains platform independence, compact instruction set, and useful abstractions such as type information and unbounded number of registers, which make it easier to analyze than machine code. Furthermore, LLVM IR comes with a large library for manipulations and optimizations.

With this approach, real-world code can be verified. The ultimate goal is to be able to verify a program without any modifications to it. This is also the goal for DIVINE [6], a well established explicit-state model checker, which aims primarily at verification of unmodified C and C++ programs with parallelism using LLVM as an intermediate representation. DIVINE aims to have full support of language features for C++, including, for example, exception handling. Furthermore, DIVINE provides near complete implementation of C and C++ standard library, including features of newest C++14 standard and the `pthread` threading library [19]. In this way, DIVINE can often be directly applied to verification of real-world code, provided it does not use inputs or platform-specific features, such as calls into the kernel of the operating system. DIVINE is able to verify wide range of properties, such as memory and assertion safety, absence of memory leaks, and liveness properties defined by linear temporal logic specification.

The other problem for practical model checking is state space explosion. The state space of all runs of a parallel program can be large, and therefore, the resources to explore it can be vastly larger than resources needed to execute the program directly. This problem is even more pronounced with verification of real-world programs, as the implemnetation details often abstracted away in translation to modelling languages are still present in them. To make verification of real-world programs feasible in more cases, DIVINE employs advanced state space reductions, including $\tau+$ and heap symmetry reductions which eliminate a large number of unnecessary interleavings [34], and tree compression to achieve memory-efficient storage of state space [36]. DIVINE also supports parallel and distributed verification [9, 4].

1.2 Relaxed Memory Models

One common source of bugs in parallel programs is the fact that modern CPUs use relaxed memory models. With relaxed memory models, the visibility of an update to a shared memory location need not be visible immediately to other threads and it might be reordered with other updates to different memory locations. This adds yet another level of difficulty to the already difficult task of programming parallel programs: memory models are hard to reason about and it is often hard to specify the desired behaviour in the programming language in question. While hardware commonly has support to ensure particular ordering of memory operations, this is often not supported by programming languages, such as older versions of C and C++. With newer programming languages, such as C11 and C++11, it is possible to specify the behaviour of the program precisely [13, 14]. Nevertheless, this is still a difficult problem, especially for high-performance tasks when it is desirable to use the weakest synchronization which is sufficient for correctness. For these reasons, it is important to be able to verify programs under relaxed memory models. This is, however, not the case for many verifiers, even if they aim at verification of real-world programs.

To further complicate the matter of relaxed memory models, the actual memory models implemented in hardware differ with CPU architectures, vendors, or even particular models of CPUs and detailed descriptions are usually not publicly available. For these reasons, it would not be practical and feasible to verify programs with regard to a particular implementation of real-world memory model. To allow analysis of programs under relaxed memory models, theoretical memory models were proposed, namely *Total Store Order* (TSO) [39] and *Partial Store Order* (PSO) [39]. Also, programming language standards, such as C++11 and LLVM, define memory models for programs written in the particular language [14, 29]. These theoretical memory models are usually described as constraints to allowed reordering of instructions which manipulate memory.

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 variety of ways to verify programs under relaxed memory models were proposed. The idea of using model checking was first discussed in the context of the Murφ model checker, which was used to generate all possible outcomes of a small, assembly-language, multiprocessor program for a given memory model [15, 27]. A technique which represents TSO store buffers with finite automata to represent possibly infinite set of its contents was introduced in [26] and later extended in [25]. In DIVINE, relaxed memory models were

first discussed in the context of the DVE modelling language [5].

More recently, a proof-of-concept support for under-approximation of Total Store Order relaxed memory model for C and C++ programs was introduced in [42]. This support is based on an LLVM transformation which automatically instruments the program-to-be-verified with store buffers, and this enriched program is verified with DIVINE. The main limitation of the transformation proposed in [42] is that it does not fully support LLVM atomic instructions with other than sequential consistency ordering and it supports only the TSO memory model.

1.3 Aims and Contributions of This Work

This work focuses on the use of LLVM transformations as a preprocessing step for verification of real-world parallel C and C++ programs using the DIVINE model checker. We demonstrate that this technique is both viable and useful, as it can aid verification of these programs. More specifically, the focus is in two areas, the first one is extension of verifiers capabilities, most importantly enriching input programs with weak memory models such that these can be verified using an unmodified model checker which assumes sequential consistency. This is a significant extension of the work presented in [42]. The new memory model instrumentation has full support for the LLVM memory model with atomic instructions, it supports verification under more relaxed memory models than total store order and it supports full range of properties supported by DIVINE. We also show a use of LLVM transformations on the case of verification of SV-COMP [12] benchmarks with DIVINE [41].

The other area is state space size reduction aided by LLVM transformations which do not change the semantics of the input program. In this area, we propose a few transformations which can be used with DIVINE.

While the techniques presented here are designed primarily for DIVINE, their nature as LLVM transformation allows their application for other model checkers, or even verifiers using different principles, provided they use LLVM as an input language and they have support for features required by these transformations.

The structure of the thesis is the following: first, in Chapter 2 we present LLVM intermediate representation and LLVM memory model and in Chapter 3 we present the architecture of DIVINE. Chapter 4 demonstrates how LLVM transformations can be used to extend the capabilities of a model checker, in particular by adding weak memory support into DIVINE as an LLVM transformation and explores the usage of LLVM transformations for state space reductions. Chapter 5 presents an experimental evaluation of the proposed techniques and finally, Chapter 6 concludes this work.

Chapter 2

LLVM

> "LLVM is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy."
>
> — LLVM Language Reference Manual [31]

LLVM [24] was originally introduced in [23] as an infrastructure for optimization. Today, LLVM is presented as compiler infrastructure, it provides programming-language-and-platform-independent tools for optimization and support for code generation for many platforms. It also defines intermediate representation — LLVM IR — a static-single-assignment-based low-level language, and a library which can be used to manipulate this intermediate representation. The name LLVM itself is often used both for the complete infrastructure as well as for LLVM IR.

LLVM IR can be represented in three ways: a human readable assembly (`.ll` file), a compact serialized bitcode (`.bc` file), or as in-memory C++ objects which can be manipulated by LLVM libraries and read from and serialized to both the other forms.

2.1 LLVM IR basics

LLVM IR human-readable representation is similar to assembly languages, but it is typed and more verbose. In this section, we will shortly describe relevant part of this human-readable LLVM representation as well as a basic structure of LLVM IR. Through the whole work, we will use `typewriter-style font` to denote a fragment of code in some programming language, most often in LLVM IR or C++. We will also use the same font for instruction and function names.

LLVM IR has two basic kinds of identifiers: global identifiers, used for global variables and functions (their names begin with `@`), and local identifiers, such as register names, types and labels (their names begin with `%`). The identifiers can be either named or unnamed, unnamed identifiers are represented using unsigned numerical values.

Modules and Functions

LLVM programs consist of *modules*. A module represents a compilation unit of the input program or a result of the LLVM linker. Modules contain functions, global variables, symbol table entries, and metadata.

A function contains a header (which defines a name, the number and type of parameters and function attributes) and a body which consists of *basic blocks*. Basic block is a continuous sequence of instructions with no branching inside, terminated by a so-called *terminator* instruction, which is an instruction which transfers control flow to another basic block (branching instruction) or exits the function (in the case of `ret` and `resume`). Each basic block has a *label* which serves as a name of the basic block. Only labels can be targets of branch instructions. Values in a function are held in *registers* which are in SSA form (they are assigned only once, at their declaration) and there is an unlimited number of them.

Most LLVM instructions operate on registers, memory manipulation is possible using only four instructions: `load`, `store`, `atomicrmw`, and `cmpxchg`. The `load` instruction loads a value of a given type from a memory location given by its pointer argument and the `store` instruction stores a value to a memory location given by its pointer argument. `atomicrmw` and `cmpxchg` are atomic instructions, they perform atomic read-modify-write and compare-and-swap. More information about these instructions can be found in Section 2.4.

Since LLVM registers are in SSA form and their address cannot be taken, they are not suitable for representation of local variables. To represent these variables, LLVM uses `alloca` instruction. `alloca` instruction takes a size and a type and returns a pointer to a memory location of given size, which will be automatically freed on function exit. Usually `alloca` is implemented using a stack when LLVM is compiled to runnable binary.

Finally, again as a consequence of SSA form, LLVM IR includes φ-nodes represented by the `phi` instruction, a special instruction which merges values from different basic blocks. `phi` instructions must be at the beginning of a basic block.

Types

LLVM is a typed language; there are primitive types, such as integral types with different bit widths (for example `i32` is a 32 bit integer, `i1` is a boolean value), floating point types (`float`, `double`), and pointer types (denoted in

the same way as in C, for example `i32*` is pointer to a 32 bit integer). Apart from primitive types, LLVM has arrays (for example `[4 x i32]` is an array of 4 integers), and structures (for example `{ i32, i8* }` is a tuple of an integer and a pointer). Furthermore, LLVM has named types and additional types, such as vector types, which are not necessary for the understanding of this work.

There are no implicit casts in LLVM, instead, a variety of of casting instructions is provided, namely `bitcast` for casting which preserves representation, `inttoptr` and `ptrtoint` to cast integers to and from pointers with same size, and `trunc`, `zext`, and `sext` for integer casts to smaller, respectively larger data types.

Metadata

LLVM modules can also contain *metadata*. Metadata are non-essential data which include additional information, for example for the compiler, optimizer or code generator. An important example of metadata are debugging informations. Metadata can be bound to LLVM instructions and functions and their names are prefixed with `!`.

2.2 LLVM Compilation Process

LLVM itself is not a complete compiler as it lacks support for translation from a higher-level programming language into LLVM IR. This translation is done by a *frontend*, such as Clang, which is a C/C++/Objective-C compiler released together with LLVM, or DragonEgg which integrates LLVM with GCC parsers and allows processing of Ada, Fortran, and others.

After the frontend generates LLVM IR, LLVM can be used to run optimizations on this IR. These optimizations are organized into *passes*; each of the passes performs a single optimization or code analysis task, such as constant propagation or inlining. LLVM passes are usually run directly by the compiler, but they can be also executed on serialized LLVM IR using the `opt` binary which comes with LLVM. Optimization passes are written in C++ using LLVM libraries.

Finally, the optimized IR has to be translated into a platform-specific assembler. This is done by a code generator, which is part of LLVM. LLVM comes with code generators for many platforms, including `x86`, `x86_64`, ARM, and PowerPC. LLVM also comes with infrastructure for writing code generators for other platforms.

2.3 Exception Handling

Exception handling in LLVM [30] is based on Itanium ABI zero-cost exception handling. This means that exception handling does not incur any overhead (such as checkpoint creation) when entering `try` blocks. Instead, all the work is done at the time exception is thrown, that is, exception handling is zero-cost until the exception is actually used.

The concrete implementation of exception handling is platform dependent and as such cannot be completely described in LLVM. It usually consists of *exception handling tables* compiled into the binary, an *unwinder* library provided by the operating system, and a language-dependent way of throwing and catching exceptions. The exception handling tables and most of the unwinder interface is not exposed into LLVM IR as it is filled in by the backend for the particular platform. Nevertheless, there must be information in LLVM IR which allows generation of this backend-specific data. For this reason, LLVM has three exception-handling-related instructions: `invoke`, `landingpad`, and `resume`.

`landingpad` is used at the beginning of an exception handling block (it can be preceded only by `phi` instructions). Its return value is a platform-and-language-specific description of the exception which is being propagated. For C++, this is a tuple containing a pointer to the exception and a *selector* which is an integral value corresponding to the type of the exception that is used in the exception catching code. The basic block which contains the `landingpad` instruction will be referred to as *landing block*.[1]

The `landingpad` instruction specifies which exceptions it can catch. It can have multiple *clauses* and each clause is either a `catch` clause, meaning the `landingpad` should be used for exceptions of a type this clause specifies, or a `filter` clause, meaning the `landingpad` should be entered if the type of the exception does not match any of the types in the clause. The type of the exception is determined dynamically, and therefore clauses contain runtime type information objects. Furthermore, a `landingpad` can be denoted as `cleanup`, meaning it should be entered even if no matching clause is found.

As the matching clause is determined at the runtime, the code in a landing block has to be able to determine which of the possible clauses (or `cleanup` flag) fired. For this reason, the return value of a `landingpad` instruction is determined using a *personality function*. Personality function is a language-specific function which is called when the exception

[1] In LLVM documentation this block is referred to as *landing pad*, however, we will use the naming introduced in [35] to avoid confusion between `landingpad` as an instruction and landing pad as a basic block which contains this instruction.

is thrown, or by the stack unwinder. For C++ and Clang, the personality function is `__gxx_personality_v0` and it returns a pointer to the exception and an integral selector which uniquely determines which catch block of the original C++ code should fire.

`invoke` instruction works similarly to the `call` instruction; it can be used to call a function in such a way that if the function throws an exception, this exception will be handled by a dedicated basic block. Unlike `call`, `invoke` is a terminator instruction, it has to be last in a basic block. Apart from the function to call and its parameters, `invoke` also takes two basic block labels, one to be used when the function returns normally and one to be used on exception propagation; the second one must be a label of a landing block.

`resume` is used to resume propagation of an exception which was earlier intercepted by an `invoke`–`landingpad` combination. The parameters are the same as returned by the `landingpad`.

It is important to note that LLVM does not have any instruction for throwing of an exceptions, this is left to the frontend to be done in language-dependent way. In C++ throwing is done by a call to `__cxa_throw` which will initiate the stack unwinding in cooperation with the unwinder library. Similarly, allocation and catching of exceptions are left to be provided by the frontend.

2.4 Memory Model and Atomic Instructions

LLVM has support for atomic instructions with well-defined behaviour in multi-threaded programs [29]. LLVM's atomic instructions are built so that they can provide the functionality required by the C++11 atomic operations library, as well as Java's volatile. Apart from atomic versions of `load` and `store` instructions, LLVM supports two atomic instructions: `atomicrmw` (atomic read-modify-write) and `cmpxchg` (atomic compare-and-exchange, also compare-and-swap) which are essentially an atomic load, immediately followed by an operation and an atomic store in such a way that no other memory action can happen between the load and the store. LLVM also contains a `fence` instruction (memory barrier) which allows for synchronization which is not part of any other operation.

2.4.1 Atomic Ordering

The semantics of these atomic instructions are affected by their *atomic ordering* which gives the strength of atomicity they guarantee. Apart from *not atomic* which is used to denote `load` and `store` instructions with no atomicity guarantee, there are six atomic orderings: *unordered*, *monotonic*, *acquire*,

release, *acquire-release*, and *sequentially consistent* (given in order of increasing strength). These atomic orderings are defined by the memory model of LLVM (which is described in detail in chapter *Memory Model for Concurrent Operation* of [31]).

In order to describe the aforementioned atomic orderings, we first need to define the *happens-before* partial order of operations of a concurrent program. Happens-before is the least partial order that is a superset of a single-thread execution order, and when *a synchronizes-with b*, it includes an edge from *a* to *b*. Synchronizes-with edges are introduced in platform-specific ways,[2] and by atomic instructions.

Unordered can be used only for `load` and `store` instructions and does not guarantee any synchronization, but it guarantees that the load or store itself will be atomic. Such an instruction cannot be split into two or more instructions or otherwise changed in a way that a load would result in a value different from all written previously to the same memory location. This memory ordering is used for non-atomic loads and stores in Java and other programming languages in which data races are not allowed to have undefined behaviour.[3]

Monotonic corresponds to `memory_order_relaxed` in the C++11 standard. In addition to guarantees given by *unordered*, it guarantees that a total ordering consistent with the happens-before partial order exists between all *monotonic* operations affecting the same memory location.

Acquire corresponds to `memory_order_acquire` in C++11. In addition to the guarantees of *monotonic* ordering, a read operation flagged as *acquire* creates a synchronizes-with edge with a write operation which created the value if this write operation was flagged as *release*. *Acquire* is a memory ordering strong enough to implement lock acquisition.

Release corresponds to `memory_order_release` in C++11. In addition to the guarantees of *monotonic* ordering, it can create a synchronizes-with edge with corresponding *acquire* operation. *Release* is memory ordering strong enough to implement lock release.

Acquire-release corresponds to `memory_order_acq_rel` in C++11 and acts as both *acquire* and *release* on a given memory location.

Sequentially Consistent corresponds to `memory_order_seq_cst` which is the default for operations on `atomic` objects in C++. In addition

[2] For example by thread creation or joining, mutex locking, and unlocking.

[3] This is in contrast with C++11 and C11 standards that specify that a concurrent, unsynchronized access to the same non-atomic memory location results in an undefined behaviour, for example `load` can return a value which was never written to a given memory location.

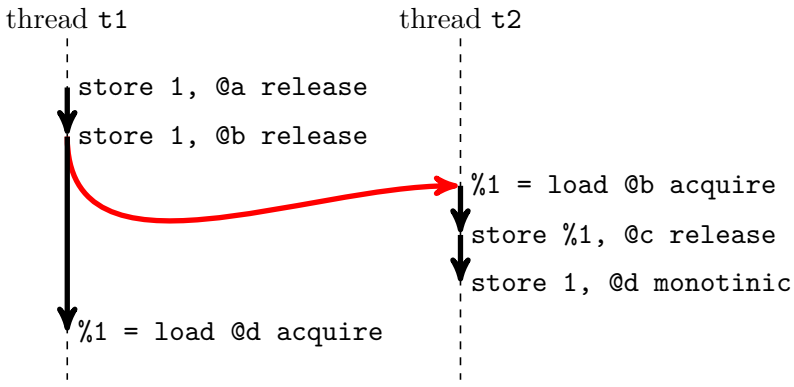


Figure 2.1: Happens-before partial order and synchronizes-with edges of a simple program with two threads (`t1` and `t2`) and global variables `a`, `b`, `c`, and `d` for an execution when first the thread `t1` executes two instructions, then `t2` executes and finally `t1` continues execution. The black arrows denote a happens-before ordering given from the single-thread execution, while the red arrow denotes a synchronizes-with edge (which is part of the happens-before partial order). Please note that there is no synchronizes-with edge between the `store` and `load` of `d` (even in case that the `load` returns the value written by the `store`) as the `store` is not *release* or stronger.

> to guarantees given by *acquire-release*, it guarantees that there is a global total order of all *sequentially-consistent* operations on all memory locations which is consistent with happens-before partial order and with modification order of all the affected memory locations.

An example of synchronizes-with edges and a happens-before partial order can be seen in Figure 2.1.

Unlike aforementioned atomic instructions, the `fence` instruction is not bound to a specific memory location. Instead, it establishes memory synchronization between *non-atomic* and *monotonic* atomic accesses. The synchronization is established if there exists a pair of `fence` instructions R and A where R is a *release* fence and A is an *acquire* fence, an atomic object M which is modified by instruction S (with at least *monotonic* ordering) after R and read by instruction L (with at least *monotonic* ordering) before A. In this case, there is a happens-before edge from R to A. Now if the read L of M observes the value written by write S, this implies that all (atomic or not) writes which happen-before the fence R also happen-before the fence A. An illustration how this can be used to implement a spin-lock can be found in Figure 2.2.

If the fence has *sequentially consistent* ordering it also participates in a global program order of all *sequentially consistent* operations. A fence is not

allowed to have *monotonic*, *unordered*, or *not atomic* ordering.

Finally, all atomic instructions can optionally have a `singlethreaded` flag which means they do not synchronize with other threads, and only synchronize with other atomic instructions within the thread. This is useful for synchronization with signal handlers.

```
1  int a;
2  std::atomic< bool > flag;
3
4  void foo() {
5      a = 42;
6      std::atomic_thread_fence( std::memory_order_release );
7      flag.store( true, std::memory_order_relaxed );
8  }
9
10 void bar() {
11     while ( !flag.load( std::memory_order_relaxed ) ) { }
12     std::atomic_thread_fence( std::memory_order_acquire );
13     std::cout << a << std::endl; // this will print 42
14 }
```

```
1  define void @_Z3foov() {
2  entry:
3    store i32 42, i32* @a, align 4
4    fence release
5    store atomic i8 1, i8* @flag monotonic, align 1
6    ret void
7  }
8
9  define void @_Z3barv() {
10 entry:
11   br label %while.cond
12
13 while.cond:
14   %0 = load atomic i8, i8* @flag monotonic, align 1
15   %tobool.i.i = icmp eq i8 %0, 0
16   br i1 %tobool.i.i, label %while.cond, label %while.end
17
18 while.end:
19   fence acquire
20   %1 = load i32, i32* @a, align 4
21   ; ...
```

Figure 2.2: An example of a use of `fence` instruction. The *release* fence (line 6 in C++, 4 in LLVM) synchronizes with the *acquire* fence (line 12 in C++, 19 in LLVM) because there exists an atomic object `flag` and an operation which modifies it with a *monotonic* ordering (lines 7, 5) after the *release* fence, and reads it, again with a *monotonic* ordering (lines 11, 14), before the *acquire* fence.

Chapter 3

DIVINE

In this chapter, we describe internal architecture of DIVINE[1] with the main focus on the implementation of LLVM verification. More details about DIVINE can be found for example in [33, 6, 10].

For the purposes of this thesis, most of the internal architecture of DIVINE is irrelevant and we will focus mostly on the LLVM interpreter, which is the only part directly involved in this work, and its interaction with the verified program is important for the understanding of the proposed LLVM transformations as well as possibility to use them outside of DIVINE. We will also describe state space reduction techniques implemented in DIVINE.

3.1 Overall Architecture

DIVINE is implemented in C++, with a modular architecture. The main modules are state space generators, exploration algorithms, and closed set stores. Currently, there are multiple implementations of each of these modules, providing different functionality. For state space generators, there are versions for different input formalisms such as LLVM, DVE [37], and UPPAAL timed automata [11]. Each of these generators define an input formalism and can be used to generate a state space graph from its input, that is, for a given state in the state space, it yields its successors and is able to report state flags. State flags are used by the exploration algorithm to detect if a goal state was reached (in the case of safety properties) or the state is accepting (in the case of LTL verification using Büchi automata).

The closed set store defines a way in which the closed set is stored, so that for a given state, it can be quickly checked if it was already seen and it is possible to retrieve algorithm data associated with this state. In DIVINE, two versions of closed set stores are present: a hash table and a hash table with lossless tree compression [36].

[1] More precisely version 3.3 which is the latest released version at the time of writing of this thesis.

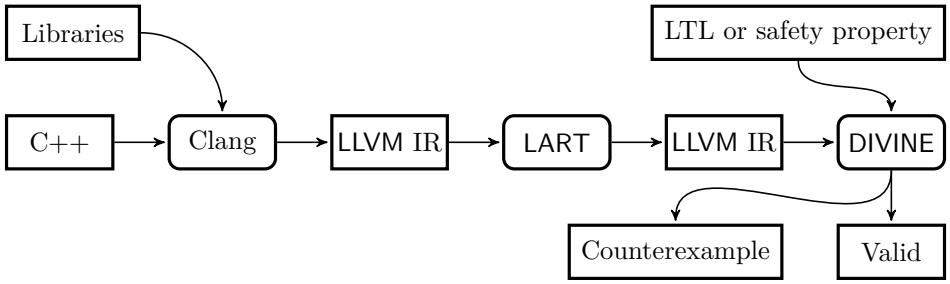


Figure 3.1: Workflow of verification of C++ programs with DIVINE. LART is optional. Boxes with rounded corners represent executables.

Finally, the exploration algorithm connects all these parts together in order to verify a given property. Which algorithm is used depends on the verified property, for safety properties, either standard BFS-based reachability or context-switch-directed reachability [40] can be used. For general LTL properties, the OWCTY algorithm [7] is used. All of these algorithms support parallel and distributed verification.

3.2 LLVM in DIVINE

LLVM support is implemented by the means of an LLVM state space generator, also referred to as an LLVM interpreter, and libraries. The interpreter is responsible for instruction execution, memory allocation and thread handling, as well as parts of exception handling. The role of the interpreter is similar to the role of the operating system and hardware for natively compiled programs. On the other hand, the libraries provide higher level functionality for user's programs, as they implement language support by the means of standard libraries for C and C++, higher-level threading by the means of the `pthread` library, and to some extent a POSIX-compatible environment, with a simulation of basic filesystem functionality. The libraries use intrinsic functions provided by the interpreter to implement low-level functionality; these functions are akin to system calls in operating systems.

We will denote all the parts implemented in LLVM bitcode, that is the libraries together with the user-provided code as the *userspace*, to distinguish it from the interpreter, which is compiled into DIVINE. Unlike the interpreter, the userspace can be easily changed, or even completely replaced without the need to modify and recompile DIVINE itself, and is closely tied to the language of the verified program, while the interpreter is mostly language-agnostic.

In order to verify programs in C or C++ in DIVINE, they are first compiled into LLVM using Clang together with libraries, the overall workflow of verification of C/C++ code is illustrated in Figure 3.1.

3.2.1 Interpreter

The LLVM interpreter is responsible for the execution of LLVM instructions and intrinsic functions (built-in operations which are represented in LLVM as calls to certain functions with a `__divine` or `llvm` prefix and are executed similarly to instructions in the interpreter). It also performs state space reductions (described in Section 3.2.5) and recognizes which states violate the verified property.

Problem Categories

DIVINE has several safety properties which can be verified in LLVM models; these properties are specified in term of problem categories. Each category is a group of related problems which should be reported as property violations. Problem categories can be reported either directly by the LLVM interpreter, or from the userspace using the `__divine_problem` intrinsic. When a problem is reported, it is indicated in the state together with the position in the program at which it was detected. Problem names are defined in the `divine/problem.h` header file, which is available to the program when it is compiled using DIVINE.

Assert corresponds to a call of `assert` function with arguments which evaluated to false.

Invalid dereference is reported by the interpreter if a load is performed from an invalid address.

Invalid argument is reported by the interpreter when a function is called with unexpected arguments, for example, non-variadic function called with more (or fewer) arguments that it expected, or an intrinsic called with wrong argument values.

Out of bounds is reported by the interpreter when an access out of the bounds of a memory object is attempted.

Division by zero is reported when integral division by zero is attempted.

Unreachable executed is reported if an `unreachable` instruction is executed. This instruction usually occurs at the end of a non-void function which lacks a return statement.

Memory leak is reported when the last pointer to a given heap memory object is destroyed before the object is freed.

Not implemented is intended to be reported by the userspace in function stubs (a function which is provided only so that bitcode does not contain undefined functions, but is not implemented, for example because it is not expected to be used).

Uninitialized is reported by the interpreter if the control flow depends on an uninitialized value.

Deadlock is reported by the userspace deadlock detection mechanisms, for example when a circular wait in `pthread` mutexes is detected.

Other is used by the userspace to report other types of problems.

Intrinsic Functions

Intrinsic functions allow the userspace to communicate with the interpreter, in order to allocate or free memory, create threads, report errors and so on. These functions are intended to be used by library writers, not by the users of DIVINE. Nevertheless, they are relevant to this work as some of them are used in proposed transformations. Since these functions are DIVINE-specific, the transformations using them would need to be modified, or equivalent functions would have to be provided should the transformation be used for other tools.

```
int __divine_new_thread( void (*entry)(void *), void *arg );
int __divine_get_tid();
```

The `__divine_new_thread` intrinsic instructs the interpreter to create a new thread, this thread will use `entry` as its entry procedure. `entry` has to accept a single `void*` argument, the interpreter will pass `arg` to the entry procedure of new thread. The function returns a thread ID used for identification of the new thread in DIVINE's interpreter. `__divine_get_tid` returns DIVINE thread ID of the thread which executed it.

When implementing threading primitives (such as those in the `pthread` library) in userspace, it is required that these are themselves free of data races. To facilitate this, DIVINE provides a way to make a section of instructs atomic; the interpreter ensures that this block of instructions is indeed executed atomically, that there is only one edge in the state space (which corresponds to the entire block of instructions, and may include any number of instructions or even function calls. It is, however, a responsibility of the library writer to use these atomic sections correctly, namely, each of these sections must always terminate; that is, there must be no (possibly) infinite cycles or recursion, such as busy-waiting for a variable to be set by another thread.

```
void __divine_interrupt_mask();
void __divine_interrupt_unmask();
```

The `__divine_interrupt_mask` function starts an atomic section; all actions performed until the atomic section ends will happen atomically. The atomic section can end in two ways, either by an explicit call to unmask function, or implicitly when function which called the mask function exits.

The behaviour of atomic sections can be more precisely explained by the means of a *mask flag* associated with each frame of the call stack. When `__divine_interrupt_mask` is called, the frame of its caller is marked with a mask flag, which can be reset by a call to `__divine_interrupt_unmask`. An instruction is part of an atomic section if the frame it corresponds to has the mask flag set. If the executed function is a function call, the

frame of the callee inherits the mask flag of the caller. However, when `__divine_interrupt_unmask` is called, it resets the mask flag of its caller, leaving mask flags of functions lower in the stack unmodified. If the current function which calls `__divine_interrupt_unmask` is not the same as the caller of `__divine_interrupt_mask`, the atomic section ends and a new atomic section will be entered when the current function returns (the caller of the current function still has the masked flag set).

```
void __divine_assert( int value );
void __divine_problem( int type, const char *data );
```

These functions can be used to report problems from the userspace. `__divine_assert` behaves much like the standard C macro `assert`: if it is called with a nonzero value, the assertion violated problem is added to the current state's problems. `__divine_problem` unconditionally reports a problem of a given category to the interpreter; the report can be accompanied by an error message passed in the `data` value.

```
void __divine_ap( int id );
```

`__divine_ap` indicates that atomic proposition represented by `id` holds in the current state. For more details on LTL in DIVINE, see Section 3.2.3.

```
int __divine_choice( int n, ... );
```

`__divine_choice` is a nondeterministic choice operator. When it is encountered, the state of the program splits into `n` copies; each copy of the state will see a different return value from `__divine_choice`, starting from 0 and up to `n` − 1. When more than one parameter is given, the choice becomes probabilistic and the remaining parameters give the probability distribution of the choices (there must be exactly `n` additional parameters). This can be used for probabilistic C++ verification, see [8] for more details.

```
void *__divine_malloc( unsigned long size );
void __divine_free( void *ptr );
int __divine_heap_object_size( void *ptr );
int __divine_is_private( void *ptr );
```

These are low-level heap access functions. `__divine_malloc` allocates a new block of memory of a given size; it never fails. `__divine_free` frees a block of memory previously allocated with `__divine_malloc`. If the block was already freed, a problem is reported. If a null pointer is passed to `__divine_free`, nothing is done.

`__divine_heap_object_size` returns the allocation size of a given object, and `__divine_is_private` returns nonzero if the pointer passed to it is private to the thread calling this function.

```
void *__divine_memcpy( void *dest, void *src, size_t count );
```

The behaviour of `__divine_memcpy` is similar to the `memmove` function in the standard C library, that is, it copies `count` bytes from `src` to `dest`; the memory areas are allowed to overlap. This intrinsic is required due to pointer tracking used for heap canonization (see [34] for details on heap canonization).

```
void *__divine_va_start();
```

This function is used to implement C macros for functions with variable number of arguments. The call to `__divine_va_start` returns a pointer to a block of memory that contains all the variadic arguments, successively assigned higher addresses going from left to right.

```
void __divine_unwind( int frameid, ... );
struct _DivineLP_Info *__divine_landingpad( int frameid );
```

These functions relate to exception handling. `__divine_unwind` unwinds all frames between the current frame and the frame denoted by `frameid`. `__divine_landingpad` gives information about the `landingpad` instruction associated with the active call in a given frame.

3.2.2 Exception Handling

In order to allow verification of unmodified programs in any programing language, it is desirable that all the language features can be handled by the verifier. When LLVM is used as an intermediate representation by the verifier, most of the language features are supported automatically by the use of a pre-existing compiler. Nevertheless, there might still be some features that require support from the verifier. C++ exceptions are such a feature and they are often omitted by verifiers for this reason.

In DIVINE, C++ exceptions are supported and the mechanisms used should allow implementation of exceptions in other programming languages entirely in the userspace, provided that they use LLVM exception handling as described in Section 2.3 and they use similar mechanisms as C++ to determine which `landingpad` clause matches the exception. The full description of DIVINE's exceptions can be found in [35].

From the point of view of a C++ program, DIVINE acts as an unwinder library, as it allows control to be transfered from the currently executing function into a landing block corresponding to an active `invoke` instruction in some stack frame deeper in the stack. The interface for this functionality is quite simple and it consists of the following functions and data types:

```
void __divine_unwind( int frameid, ... );
```

```
struct _DivineLP_Clause {
    int32_t type_id;
    void *tag;
};

struct _DivineLP_Info {
    int32_t cleanup;
    int32_t clause_count;
    void *personality;
    struct _DivineLP_Clause clause[];
};

struct _DivineLP_Info *__divine_landingpad( int frameid );
```

`__divine_unwind` unwinds all frames between the current frame and the frame denoted by `frameid`. No landing pads are triggered in the intermediate frames, if there is a `landingpad` for the active call in the frame in which the unwinding ends and any arguments other than `frameid` were passed to `__divine_unwind`, this landing pad returns arguments passed to `__divine_unwind` (if the active call instruction in the destination frame is `call` and not `invoke`, the extra arguments are returned as the result of the function). The `frameid` is 0 for the caller of `__divine_unwind`, -1 for its caller and so on.

`__divine_landingpad` gives information about `landingpad` associated with the active `invoke` in the frame denoted by `frameid`, if there is some. It returns a pointer to a `_DivineLP_Info` object which corresponds to the landing pad, or a null pointer if the frame does not exist. If the active instruction in the target frame is a `call` instead of an `invoke`, the returned `_DivineLP_Info` object will contain no clauses. The returned structure encodes information about the `landingpad` it corresponds to and about the personality function used by its enclosing function. There is a flag which indicates whether the landing block is a cleanup block (it should be entered even if the exception does not match any of the clauses), and an array of `_DivineLP_Clause` structures which encodes the clauses of the `landingpad`. For each of these clauses, there is an identifier which should be returned as a selector from the `landingpad` if this clause is matched, and a pointer to a language-specific `tag` (which is a type information object in C++).

Using these functions, a function which throws an exception can be implemented: it goes through the stack asking for `_DivineLP_Info` in each frame beginning with its caller, and for each of them checks if the exception type matches any of the clauses in the `landingpad`. When a matching clause is found, the corresponding type id is set in the exception object, which is then passed into a `personality` function. The personality function returns a value which should be returned from the `landingpad` instruction, so this value is

```
#include <divine.h>

enum APs { c1in, c1out, c2in, c2out };
LTL(exclusion,
    G((c1in -> (!c2in W c1out)) && (c2in -> (!c1in W c2out))));

void critical1() {
    AP( c1in );
    AP( c1out );
}

void critical2() {
    AP( c2in );
    AP( c2out );
}
```

Figure 3.2: A fragment of C program which uses LTL property `exclusion` to verify that functions `critical1` and `critical2` cannot be executed in parallel.

passed, together with the frame id of the target frame, into `__divine_unwind` to perform the unwinding.

The `resume` instruction implementation is in the interpreter. It finds the nearest `invoke` in the call stack and transfers control to its `landingpad` which will return the value passed to the `resume`.

Apart from the aforementioned exception handling `__divine_unwind` is also usable for the implementation of functions such as `pthread_exit`. In this case, the stack is fully unwound, which causes the thread to terminate. Furthermore, [35] presents a minor extension of the exception handling mechanism which would allow an implementation of the `setjmp`/`longjmp` POSIX functions; this extension is not implemented in DIVINE.

3.2.3 LTL

LTL support in DIVINE is implemented using an explicit set of atomic propositions defined as `enum APs` in the verified program. These atomic propositions are activated explicitly using a macro named `AP` (which uses `__divine_ap` internally), and they are active in the state where `AP` is called. As a result of this explicit activation of atomic propositions, it is not possible for more than one atomic proposition to be true in any state, and it is not possible to express certain formulas, namely $G(a)$ for $a \in$ `APs`. The LTL properties which should be verified are encoded in the program using a macro named `LTL`. See Figure 3.2 for an example of a program with LTL in DIVINE.

3.2.4 Userspace

DIVINE has userspace support for C and C++ standard libraries using PDCLib and libc++. This support is mostly complete, the most notable missing parts are locale support (which is missing in PDCLib) and limited support for filesystem primitives (there is support for the creation of directory snapshots which can be accessed and processed using standard C, C++, or POSIX functions).

Apart from standard libraries, DIVINE provides the `pthread` threading library, which provides thread support for C and older versions of C++ which do not include thread support in the standard library and is also used as the underlying implementation of C++11 threads. Furthermore, there is rudimentary support for POSIX-compatible filesystem functions, including certain types of UNIX domain sockets; however, this library is still under development at the time of writing of this thesis.

From the point of view of this thesis, all the userspace is considered to be part of the verified program; that is, any LLVM transformation runs on the entire userspace, not just the parts provided by the user of DIVINE.

3.2.5 Reduction Techniques

In order to make verification of real-world LLVM programs tractable, it is necessary to employ state space reductions. DIVINE uses $\tau+$ reduction to eliminate unnecessary intermediate states and heap symmetry reduction when verifying LLVM [34]. These reductions preserve all safety and LTL properties which can be expressed in DIVINE. Furthermore, DIVINE uses lossless, modeling-language-agnostic tree compression of the entire state space [36].

$\tau+$ Reduction

In LLVM, many instructions have no effect which could be observed by threads other than the one which executes the instruction. This is true for all instructions which do not manipulate memory (they might still use registers, which are always private to the function in which they are declared), or which manipulate memory which is thread private.

DIVINE uses this observation to reduce the state space. It is possible to execute more than one instruction on a single edge in the state space, provided that only one of them has an effect visible to other threads (is *observable*). To do this, the interpreter tracks if it has executed any observable instruction and emits a state just before another observable instruction is executed (this, of course, is suppressed in atomic sections, where only tracking takes place; a state can be emitted only after the end of the atomic section). To decide which instructions are observable, DIVINE uses the following heuristics:

- any instruction which does not manipulate memory is not observable

(that is, all instructions apart from `load`, `store`, `atomicrmw`, `cmpxchg` and the built-in function `__divine_memcpy`[2]);
- for the memory-manipulating instructions, it is checked whether the concerned memory location can be visible by other threads; if it can, the instruction is observable.

To detect which memory can be accessed from particular threads, DIVINE checks the reachability of a given memory object in the memory graph (memory objects are nodes, pointers are edges of this graph). In order to check if thread a has access to a memory object x, it has to check that x is reachable either from global variables or from registers in any stack frames which belong to a. To build the memory graph, DIVINE remembers which memory locations contain heap pointers (this is required as it is valid to cast a pointer to and from a number in both LLVM and C++).

However, in order to ensure that successor generation terminates, it is necessary to avoid execution of infinite loops (or recursion) on a single edge in the state space (this could happen, for example, due to an infinite cycle of unobservable instructions). For this reason, DIVINE also tracks which program counter values were encountered during successor generation, and if any of them is to be encountered for a second time, a state is emitted before the second execution of given instruction.

3.3 LART

LART is a tool for LLVM transformation and optimization developed together with DIVINE; it was first introduced in [33] as a platform for implementation of static abstraction and refinement of LLVM programs. It is intended to integrate LLVM transformations and analyses in such a way that would make it easy to implement new and reuse existing analyses.

Before the time of writing of this thesis, LART was never released and it contained few mostly incomplete analyses and a proof-of-concept version of an LLVM transformation which adds weak memory model verification support to existing LLVM program (this part was presented in [42]). Most of the work presented in this thesis is implemented in LART.

[2] In fact DIVINE 3.3 does not consider `__divine_memcpy` observable, this is a bug discovered and fixed during the writing of this thesis.

Chapter 4

Proposed LLVM Transformations

In this chapter, we will propose LLVM transformations which aim at improving model checking capabilities and reduce state-space size. Most of the proposed transformations were implemented in LART and will be released together with the next release of DIVINE.

4.1 Extensions to DIVINE

In order to implement some of the proposed transformations, it was necessary to perform minor changes to the LLVM interpreter in DIVINE. All these changes are implemented in the version of DIVINE submitted with this thesis and are described in this section.

4.1.1 Simplified Atomic Masks

The original semantics of `__divine_interrupt_mask` were not well suited for composition of functions which use it. For this reason, we reimplemented this feature so that it behaves as if `__divine_interrupt_mask` locks a global lock and `__divine_interrupt_unmask` unlocks it, and we devised a higher-level interface for this feature. This interface is described in Section 4.3.

4.1.2 Assume Intrinsic

```
void __divine_assume( int value );
```

We extended DIVINE with a new intrinsic function which implements the well-known assume statement. If `__divine_assume` is executed with a zero `value`, it stops the interpreter and causes it to ignore the current state. `__divine_assume` is useful for implementation of synchronization primitives, for example in weak memory model simulation (see Section 4.4).

This function should be used primarily by DIVINE developers; it combines well with atomic masks to create conditional transitions in the state space.

4.1.3 Extended State Space Reductions

When evaluating which transformations are useful for state space reductions, we identified several cases in which a runtime solution by extension of the $\tau+$ reduction was more efficient. For this reason, the existing reduction technique was improved and these improvements are implemented in the version of DIVINE submitted with this thesis. Please refer to Section 3.2.5 for details about $\tau+$ reduction. The evaluation of the impact of the proposed changes to $\tau+$ reduction can be found in Section 5.1.

Control Flow Cycle Detection

First, we improved upon the overly pessimistic control flow cycle detection heuristic. This detection is used to make sure that successor generation terminates and it is based on detection of repeating program counter values. However, the set of encountered program counter values was originally reset only at the beginning of state generation. For this reason, it was not possible to execute one function more that once on one edge in the state space as the program counter of this function was already in the set of seen program counters on the second invocation. Therefore, a new state was generated before the function could be executed for a second time, which resulted in unnecessary states.

To alleviate this limitation, all program counter values of a function are deleted from the set of seen program counter values every time the function exits. This way, two consecutive calls to the same function need not generate a new state, while a call in the loop will generate a new state before the second invocation (since the `call` instruction repeats), and recursion will also generate a new state at the second entry into the recursive function.

This improved reduction is now enabled by default. The original behaviour can be obtained by option `--reduce=tau+,taustores` to `divine verify` (the extended reduction can be explicitly enabled by `tau++` key in the `--reduce` option if necessary).

Independent Loads

Another case of overly strict reduction heuristic are independent loads from shared memory locations. Consider two shared memory locations (for example shared variables) a and b such that $a \neq b$. The proposition is that we can extend $\tau+$ reduction in such a way that load from a and load from b can be performed without an intermediate state (that is, on a single edge in the state space). We will now show correctness of this proposition.

Suppose thread $t1$ performs a load of a and then a load of b (and there are no actions which would be considered observable by $\tau+$ in-between).

- If any other thread performs a load of a or b, this clearly does not interfere with $t1$.
- If some other thread $t2$ writes[1] into a, this write is always an observable action and it can happen either

  a) before the load of a by $t1$ or after the load of b by $t1$; in these cases, the proposed change has no effect;
  b) after the load of a, but before the load of b by $t1$; this case is not possible with the extended reduction, but an equivalent result can be obtained if a is written after the load of b, as this load is independent and therefore its result does not depend on the value of a.

- If some other thread $t2$ writes into b, this write is always an observable action and it can happen either

  a) before the load of a by $t1$ or after the load of b by $t1$; in these cases, the proposed change has no effect;
  b) after the load of a, but before the load of b by $t1$; again, this case is not possible with the extended reduction, but an equivalent result can be obtained if b is written before the load of a (it does not change its result as $a \neq b$).

- There can be no synchronization which would disallow any of the aforementioned interleavings as thread $t2$ cannot detect where in the sequence of instructions between load a and load b thread $t1$ is (there are no visible actions between the loads).
- On the other hand, if there are any other visible actions between these loads, or if $a = b$, the conditions are not met and the loads are not performed atomically.

The same argument can be applied to more than two independent loads from a single thread; this way, any sequence of independent loads and unobservable actions can execute atomically.

Furthermore, the reduction can be extended to a sequence of independent loads followed by a write into a memory location distinct from all the memory locations of the loads. The argument is similar to the argumentation for the case of a sequence of loads. If a write w from another thread happens between

[1] Write can be implemented using `store`, `atomicrmw`, or `cmpxchg` instructions, or by `__divine_memcpy` intrinsic.

the loads and the write w' in the sequence, a write with the same effect can happen in the reduced state space too: if w and w' write to different memory locations than w can happen after the sequence which ends with w'; otherwise, all the loads in the sequence are independent of w and therefore w can happen before the sequence.

To implement this reduction, DIVINE now tracks which memory objects were loaded while it generates a state. If a memory object is loaded for the first time, its address is saved and this load is not considered to be observable. If the same object is to be accessed for the second time during generation of the state, the state is emitted just before this access. If a non-private object is to be loaded after a new value was stored into it, a state is emitted before this load too. This reduction is now enabled by default; the original behaviour can be obtained by using the option `--reduce=tau++,taustores` to `divine verify` (the extended reduction can be explicitly enabled by the `tauloads` key in the `--reduce` option).

4.2 Analyses and Transformation Building Blocks

Many tasks done in LLVM transformations are common and, therefore, should be provided as separate and reusable analyses or transformation building blocks, so that they can be readily used when required and it is not necessary to implement them ad-hoc every time. In some cases (for example dominator tree and domination relation), analyses are provided in the LLVM library, and LLVM also provides useful utilities for instruction and basic block manipulation, such as basic block splitting and instruction insertion. In other cases, it is useful to add to this set of primitives, and, for this reason, LART was extended to include several such utilities.

4.2.1 Fast Instruction Reachability

While LLVM has support for checking whether the value of one instruction might reach some other instruction (using the `isPotentiallyReachable` function), this function is slow if many-to-many reachability is to be calculated (this function's time complexity is linear with respect to the number of basic blocks in the control flow graph of the function). For this reason, we introduce an analysis which pre-calculates the reachability relation between all instructions and allows fast querying; this analysis can be found in `lart/analysis/bbreach.h`.

To calculate instruction reachability quickly and store it compactly, we store the transitive closure of basic block reachability instead; the transitive closure of instruction reachability can be easily retrieved from this information. Instruction i other than `invoke` reaches instruction j in at least one step if and only if the basic block $b(i)$ of instruction i reaches in at least one step the

basic block $b(j)$ of instruction j or if $b(i) = b(j)$ and i is earlier in $b(i)$ than j. For the `invoke` instruction, the situation is more complicated as it is the only terminator instruction which returns a value, and its value is available only in its normal destination block and not in its unwind destination block (the landing block which is used when the function called by the `invoke` throws an exception). For this reason, the value of `invoke` instruction i reaches instruction j if and only if $b(j)$ is reachable (in any number of steps, including zero) from the normal destination basic block of i.

Basic block reachability is calculated in two phases, first the basic block graph of the function is split into strongly connected components using Tarjan's algorithm. This results in a directed acyclic graph of strongly connected components. This SCC collapse is recursively traversed and the transitive closure of SCC reachability is calculated.

The theoretical time complexity of this algorithm is linear in the size of the control flow graph of the function (which is in the worst case $\mathcal{O}(n^2)$ where n is the number of basic blocks). In practice, associative maps are used in several parts of the algorithm, resulting in the worst case time complexity in $\mathcal{O}(n^2 \cdot \log n)$ for transitive closure calculation and $\mathcal{O}(\log n)$ for retrieval of the information whether one block reaches another. However, since in practice control flow graphs are sparse,[2] the expected time complexity is $\mathcal{O}(n \log n)$ for transitive closure calculation.

4.2.2 Exception Visibility

Often, LLVM is transformed in a way which requires that certain cleanup action is performed right before a function exits; one such example would be unlocking atomic sections, used in Section 4.3. Implementing this for languages without non-local control flow transfer other than with `call` and `ret` instructions, for example standard C, would be fairly straightforward. In this case, it is sufficient to run the cleanup just before the function returns. However, while pure standard-compliant C has no non-local control transfer, in POSIX there are `setjmp` and `longjmp` functions which allow non-local jumps and, even more importantly, C++ has exceptions in its standard. Since `longjmp` and `setjmp` are not supported in DIVINE, we will assume they will not be used in the transformed program. On the other hand, exceptions are supported by DIVINE and, therefore, should be taken into account.

In the presence of exceptions (but without `longjmp`); a function can be exited in the following ways:

- by a `ret` instruction;

[2] The argument is that all terminator instructions other that `switch` have at most two successors and `switch` is rare, for this reason, the average number of edges in control flow graph with n basic blocks is expected to be less than $2n$.

- by a `resume` instruction which resumes propagation of an exception which was earlier intercepted by a `landingpad`;
- by an explicit call to `__divine_unwind`;
- when an exception causes unwinding, and the active instruction through which the exception is propagating is a `call` and not an `invoke`, or it is an `invoke` and the associated `landingpad` does not catch exceptions of given type; in this case, the frame of the function is unwound and the exception is not intercepted.

The latest case happens often in C++ functions which do not require any destructors to be run at the end of the function. In those cases, Clang usually generates a `call` instead of an `invoke` even if the callee can throw an exception, as it is not necessary to intercept the exception in the caller. Also, if the function contains a `try` block, Clang will generate an `invoke` without a `cleanup` flag in the `landingpad` as there is no need to run any destructors. The problem with the last case is that the function exit is implicit: it is possible at any `call` instruction which can throw, or at an `invoke` with a `landingpad` without a `cleanup` flag.

In order to make it possible to add code at the end of the function, it is therefore necessary to eliminate this implicit exit without interception of the exception. The transformation must be performed in such a way that it does not interfere with exception handling which was already present in the transformed function.

Therefore, we need to transform any call in such a way that if the called function can throw an exception, it is always called by `invoke`, and all the `langingpad` instructions have a `cleanup` flag. Furthermore, this transformation must not change the observable behaviour of the program. If an exception would fall through without being intercepted in the original program, it needs to be intercepted and immediately resumed, and if the exception was intercepted by the original program, its processing must be left unchanged (while the fact that the exception is intercepted by a `langingpad` and immediately resumed makes the run different from the run in the original program, this change is not distinguishable by any safety or LTL property supported by DIVINE, and therefore the transformed program can be considered equivalent to the original).

After this transformation, every exception is visible in every function it can propagate through. Now if we need to add cleanup code to the function, it is sufficient to add it before every `ret` and `resume` instruction and before calls to `__divine_unwind`, as there is no other way the function can be exited.

If `setjmp/longjmp` were implemented as an extension of exception handling support as described in [35], it would require minor modification of this transformation. It would be necessary to run transformation cleanups, but not cleanups done by higher level language (such as C++ destructors) when unwinding is caused by `longjmp` (`longjmp` is not required to trigger

destructors in C++; in fact, it is undefined behaviour to cause unwinding of any function with nontrivial destructors by `longjmp`). To achieve this, a fresh selector ID for `longjmp` would be assigned and a `catch` clause corresponding to this ID would be added to each `langingpad`. If this clause is triggered, only transformation cleanups would be run before the unwinding would be resumed.

Implementation

The idea outlined above is implemented in `lart/support/cleanup.h` by the function `makeExceptionsVisible`. Any `call` instruction for which we cannot show that the callee cannot throw an exception is transformed into an `invoke` instruction, which allows us to branch out into a landing block if an exception is thrown by the callee. The `landingpad` in the landing block needs to be set up so that it can catch any exception (this can be done using the `cleanup` flag for `landingpad`). The instrumentation can be done as follows:

- for a call site, if it is a `call`:

  1. given a `call` instruction to be converted, split its basic block into two just after this instruction (we will call these blocks *invoke block* and *invoke-ok block*);
  2. add a new basic block for cleanup; this block will contain a `landingpad` instruction with a `cleanup` flag and no `catch` clauses and a `resume` instruction (we will call this block *invoke-unwind block*);
  3. replace the `call` instruction with an `invoke` of the same function and with the same parameters, its normal destination is set to the invoke-ok block and its unwind destination is set to invoke-unwind block;

- otherwise, if it is an `invoke` and its unwind block does not contain the `cleanup` flag in the `landingpad`:

  1. create a new basic block which contains just a resume instruction (*resume block*)
  2. add `cleanup` flag into the `landingpad` of the unwind block of the `invoke` and branch into the *resume block* if the landing block is triggered due to `cleanup` (selector value is 0),

- otherwise leave the instruction unmodified.

Any calls using the `call` instruction with a known destination which is a function marked with `nounwind` will not be modified. Functions marked with `nounwind` need not be checked for exceptions since LLVM states that

these functions should never throw an exception and therefore we assume that throwing an exception from such a function will be reported as an error by the verifier.

After the call instrumentation, the following holds: every time a function is entered during stack unwinding due to an active exception, the control is transfered to a landing block. Moreover, if before this transformation the exception would not have been intercepted by `landingpad` in this function, after the transformation the exception will be rethrown by `resume`.

Furthermore, if the transformation adds a `landingpad` into a function which did not contain any `landingpad` before, it is necessary to set a personality function for this function. For this reason, the personality function which is used by the program is a parameter of the transformation.

An example of the transformation can be seen in Figure 4.1.

Finally, to simplify transformations which add cleanups at function exits, a function `atExits` is available in the same header file.

4.2.3 Local Variable Cleanup

A special case of cleanup code ran before a function exits is local variable cleanup, cleanup code which needs to access local variables (results of `alloca` instructions). One of the transformations which requires this kind of cleanup is the transformation to enable weak memory model verification (Section 4.4.5), while another case can arise from compiled-in abstractions proposed in [33]. Variable cleanups are essentially akin to C++ destructors, in a sense that they get executed at the end of the scope of the variable, no matter how this happens (with the possible exception of thread termination).

The local variable cleanup builds on top of the function cleanups described in Section 4.2.2. Unlike the previous case, it is not necessary to transform all calls which can throw an exception; it is sufficient to transform calls which can happen after some local variable declaration (that is, a value of an `alloca` instruction can reach a `call` or an `invoke` instruction). After this transformation, cleanup code is added before every exit from the function. However, in order for the cleanup code to work, it needs to be able to access all local variables which can be defined before the associated function exits (results of all `alloca` instructions from which this exit can be reached). This might not be always be the case in the original program, see Figure 4.2 for an example. In this example, `%y` is defined in the `if.then` basic block and it needs to be cleared just before the `return` instruction at the end of the `if.end` basic block and the definition of `%y` does not dominate the cleaning point.

The cleanup cannot be, in general, inserted after the last use of a local variable as the variable's address can escape the scope of the function and even the thread in which it was created and, therefore, it is not decidable

```
void foo() { throw 0; }
void bar() { foo(); }
int main() {
    try { bar(); }
    catch ( int & ) { }
}
```

An example of a simple C++ program which demonstrates use of exceptions, the exception is thrown by `foo`, goes through `bar` and is caught in `main`.

```
define void @_Z3barv() #0 {
entry:
  call void @_Z3foov()
  unreachable
}
```

LLVM IR for function `bar` of the previous example (the names of functions are mangled by the C++ compiler). It can be seen that while `foo` throws an exception and this exception propagates through `bar`, `bar` does not intercept this exception in any way.

```
1  define void @_Z3barv() #0 personality
2      i8* bitcast (i32 (...)* @__gxx_personality_v0 to i8*) {
3  entry:
4    invoke void @_Z3foov()
5        to label %fin unwind label %lpad
6  lpad:
7    %0 = landingpad { i8*, i32 } cleanup
8    resume { i8*, i32 } %0 ; rethrow the exception
9  fin:
10   unreachable
11 }
```

A transformed version of `bar` in which the exception is intercepted and, therefore, visible in this function, and it is immediately resumed. Cleanup code would be inserted just before line 8. The original basic block `entry` was split into `entry` and `fin` and the `call` instruction was replaced with an `invoke` which transfers control to the `lpad` label if any exception is thrown by `foo`. The function header is now extended with a personality function and this personality function calculates the value returned by `landingpad` for a given exception.

Figure 4.1: An example of the transformation of a function which makes exceptions visible.

```
entry:
  %x = alloca i32, align 4
  store i32 1, i32* %x, align 4
  %0 = load i32, i32* %x, align 4
  %cmp = icmp eq i32 %0, 0
  br i1 %cmp, label %if.then, label %if.end

if.then: ; preds = %entry
  %y = alloca i32, align 4
  store i32 1, i32* %y, align 4
  br label %if.end

if.end:  ; preds = %if.then, %entry
  ; cleanup will be inserted here
  ret i32 0
```

Figure 4.2: An example of LLVM code in which a local variable is allocated in only one branch and therefore does not dominate the function exit. While Clang usually moves all `alloca` instructions into the first block of the function, the example is still a valid LLVM bitcode, and therefore should be handled properly.

```
if.end:  ; preds = %if.then, %entry
  %y.phi = phi i32* [ null, %entry ], [ %y, %if.then ]
  ; cleanup will be inserted here, it will access %y.phi
  ret i32 0
```

Figure 4.3: Transformation of the last basic block from Figure 4.2 to allow cleanup of `%y`.

when its scope ends. Nevertheless, it is safe to insert the cleanup just before the function exits as the variable will cease to exist when the function exits, that is, immediately after the cleanup.

To make all local variables which can reach an exit point of a function accessible at this exit point, we will first insert φ-nodes in such a way that any `alloca` is represented in any block which it can reach, either by its value if the control did pass the `alloca` instruction (the local variable is defined at this point), or by the `null` constant if the control did not pass the `alloca`. For our example, the result of the modification is shown in Figure 4.3. In this code, `%y.phi` represents `%y` at the cleanup point. It can either be equal to `%y` if the control passed through the definition of `%y`, or `null` otherwise.

While this transformation changes the set of runs of a program, all the runs in the original program have equivalent (from the point of view of

safety and LTL properties supported by DIVINE) runs in the transformed program. The only difference is that there can be some intermediate states (which correspond to the cleanup) in the transformed program's runs. This is, however, not distinguishable in DIVINE unless the cleanup code signals a problem or sets an atomic proposition.

Implementation

To calculate which `alloca` instructions can reach a function exit, a version of the standard reaching definitions analysis [1] is used. Using this analysis, we compute which `alloca` instruction values reach the end of each basic block of the function and for every such value which does not dominate the end of the basic block, a φ-node is added. For each basic block, the algorithm also keeps track of the value which represents a particular `alloca` instruction in this basic block (it can be either the `alloca` itself, or a `phi` instruction). These values are passed to the cleanup code. The transformation is done by the `addAllocaCleanups` function which is defined in `lart/support/cleanup.h`.

4.3 New Interface for Atomic Sections

The interface for atomic sections in the verified code (described in Section 3.2.1) is hard to use, the main reason being that while the mask set by `__divine_interrupt_mask` is inherited by called functions, these functions have no way of knowing if an instruction executes inside an atomic section, and therefore, a callee can accidentally end the atomic section by calling `__divine_interrupt_unmask`. This is especially bad for composition of atomic functions, see Figure 4.4 for an example. For this reason, the only compositionally safe way to use the DIVINE's original atomic sections is to never call `__divine_interrupt_unmask` and let DIVINE end the atomic section when the caller of `__divine_interrupt_mask` ends.

To alleviate the aforementioned problems, we reimplemented atomic sections in DIVINE. The new design uses only one *mask flag* to indicate that the current thread of execution is in an atomic section; this flag is internal to the interpreter and need not be saved in the state (indeed, it would be always set to false in the state emitted by the generator, because the state can never be emitted in the middle of an atomic section). Furthermore, we modified `__divine_interrupt_mask` to return an `int` value corresponding to the value of *mask flag* before it was set by this call.

To make the new atomic sections easier to use, we provide higher level interfaces for atomic sections in the form of a C++ library and annotations. The C++ interface is intended to be used primarily by developers of the language support and libraries for DIVINE, while the annotations are designed to be used by users of DIVINE.

```
1  void doSomething( int *ptr, int val ) {
2      __divine_interrupt_mask();
3      *ptr += val;
4      __divine_interrupt_unmask();
5      foo( ptr );
6  }
7
8  int main() {
9      int x = 0;
10     __divine_interrupt_mask();
11     doSomething( &x );
12     __divine_interrupt_unmask();
13 }
```

Figure 4.4: An example of a composition problem with the original version of DIVINE's atomic sections. The atomic section begins on line 10 and is inherited by `doSomething`. The atomic section ends by the unmask call at line 4 and the rest of `doSomething` and `foo` are not executed atomically. The atomic section is then re-entered when `doSomething` returns.

The C++ interface is RAII-based,[3] and works similarly to the C++11 mutex ownership wrapper `unique_lock` with a recursive mutex. An atomic section begins by construction of an object of the type `divine::InterruptMask` and it is left either by a call to the `release` method of this object or by the destructor of the object. When atomic sections are nested, only the `release` on the object which started the atomic section actually ends the atomic section. See Figure 4.5 for an example.

The annotation interface is based on a LART transformation pass and annotations which can be used to mark an entire functions as atomic. A function can be marked atomic by adding `__lart_atomic_function` to the function header, see Figure 4.6 for an example. While this is a safer way to use atomic sections than explicitly using `__divine_interrupt_mask`, it is still necessary that the atomic function always terminates (e.g. does not contain an infinite cycle).

Implementation of Annotation Interface

Atomic sections using annotations are implemented in two phases. First, the function is annotated with `__lart_atomic_function` which is a macro which expands to GCC/Clang attributes `annotate("lart.interrupt.masked")`

[3] Resource Acquisition Is Initialization, a common pattern in C++. A resource is allocated inside an object and safely deallocated when that object's scope ends, usually at the end of a function in which the object was declared [32].

```
#include <divine/interrupt.h>

void doSomething( int *ptr, int val ) {
    divine::InterruptMask mask;
    *ptr += val;
    // release the mask only if 'mask' object owns it:
    mask.release();
    // masked only if caller of doSomething was masked:
    foo( ptr );
}

int main() {
    int x = 0;                      // not masked
    divine::InterruptMask mask;
    doSomething( &x );              // maksed
    x = 1;                          // still masked
    // mask ends automatically at the end of main
    // (if it began here)
}
```

Figure 4.5: An example use of the C++ interface for the new version of atomic sections in DIVINE.

```
#include <lart/atomic.h> // defines the annotation
// this function executes atomically
int atomicInc( int *ptr, int val ) __lart_atomic_function {
    int prev = *ptr;
    *ptr += val;
    return prev;
}
```

Figure 4.6: An example of using the annotation interface for atomic functions in DIVINE. The function `atomicInc` is aways executed atomically and it is safe to execute it inside another atomic section.

and `noinline`; the first attribute is used so that the annotated function can be identified in LLVM IR, the second to make sure the function will not be inlined.

The second phase is the LART pass which adds atomic sections into annotated functions. The pass implementation can be found in class `Mask` in `lart/reduction/interrupt.cpp`. For each function which is annotated, it adds a call to `__divine_interrupt_mask` at the beginning of the function, and a call to `__divine_interrupt_unmask` before any exit point of the function (using the cleanup transformation introduced in Section 4.2.2). The unmask call is conditional: it is only called if the mask call returned 0 (that is, the current atomic section begun by this call).

This LART pass was integrated into the program build with the `divine compile` command and, therefore, it is not necessary to run LART manually to make atomic sections work.

4.4 Weak Memory Models

In [42], it was proposed to add weak memory model simulation using LLVM transformation. In this section, we will present an extended version of this transformation. The new version supports the LLVM memory model fully, including support for atomic instructions, support for more relaxed memory models (than total store order), and specifying the memory model to use as a parameter of the transformation. It also allows for verification of the full range of properties supported by DIVINE (the original version was not usable for verification of memory safety). Furthermore, we propose ways to reduce the state space size compared to the original version. The evaluation of the proposed transformation can be found in Section 5.2.

4.4.1 Representation of the LLVM Memory Model Using Store Buffers

Relaxed memory models can be simulated using store buffers. Any write is first done into a thread-private buffer and therefore it is invisible for other threads. This buffer keeps the writes in FIFO order and it can be flushed nondeterministically into memory; the order of flushing depends on the particular memory model. For total store order, only the oldest entry can be flushed, for partial store order any entry can be flushed, provided that there is no older entry for the same memory location. Furthermore, any load has to first look into the store buffer of its thread for newer values of the loaded memory location and only if there is no such value, it can look into memory. See Figure 4.7 for an example of store buffer instrumentation.

The basic idea behind the proposed LLVM memory model simulation is that a store buffer can be flushed nondeterministically in any order, even though not all orders result in valid runs of the program. The store buffer

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

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

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

In this example, each of the threads first writes into a global variable and later it 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 can proceed. However, under total store order, $x = 0, y = 0$ is also possible: this corresponds to the reordering of the load on line 3 before the independent store on line 2. This behaviour can be simulated using store buffers; in this case the store on line 2 is not immediately visible, it is done into a store buffer. The following diagram shows a (shortened) execution of the listed code. Dashed lines represent where the given value is read from/stored to.

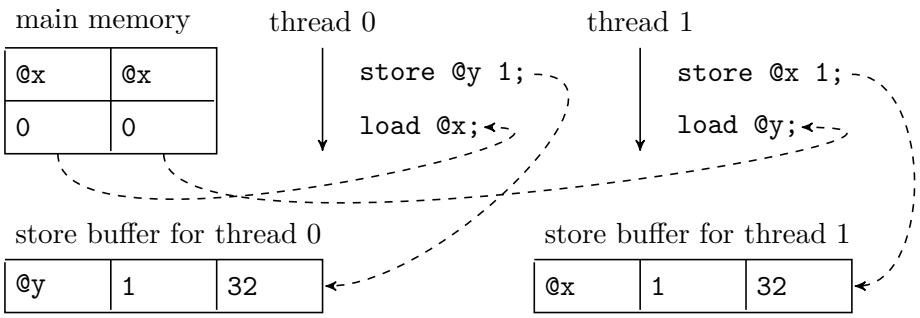


Figure 4.7: An illustration of a behaviour which is not possible with sequential consistency. It is, however, possible with total store order or any more relaxed memory model.

entries are enriched with information about the instruction which created them and therefore the validity of a particular run can be checked when a load, a read fence, or an atomic instruction is performed, and the invalid runs are discarded (using `__divine_assume`).

The approximation uses store buffers to delay `store` and `fence` instructions. There is a bounded store buffer associated with each thread of the program and this buffer is filled by `store` and `fence` instructions and flushed nondeterministically. The store buffer contains *store entries*, each of them created by a single `store` instruction and contains the following fields:

- the **address** of the memory location of the store,
- the **value** of the store,
- the **bit width** of the stored value (the value size is limited to 64 bits),
- the **atomic ordering** used by the store,
- a bit which indicates if the value **was already flushed** (*flushed flag*),
- a bit set of **threads which observed the store** (*observed set*).

Apart from store entries, a store buffer can contain *fence entries* which correspond to `fence` instructions with at least *release* ordering (write fence). Fence entries have following fields:

- the **atomic ordering** of the fence,
- a bit set of **threads which observed the fence**.

Store buffer entries are saved in the order of execution of their corresponding instructions.

Atomic instructions are not directly represented in store buffers; instead, they are split into their non-atomic equivalents using `load` and `store` instructions which are performed atomically in a DIVINE's atomic section and transformed using weak memory model. Finally, `load` instructions and read fences have constraints on the state of store buffers in which they can execute. These constraints ensure that the guarantees given by the atomic ordering of the instruction are met.

The aim of the proposed transformation is to approximate the LLVM memory model as closely as possible (except for the limitations given by the bound on store buffer size). For this reason, we support all atomic orderings apart from *not atomic*, which is modelled as *unordered*.[4] The store buffer is

[4]The difference between *not atomic* and *unordered* is that both the compiler and the CPU is allowed to split *not atomic* operations and the value of a concurrently-written *not atomic* location is undefined while for an *unordered* operation, it is guaranteed to be one of the previously written values; however, on most modern hardware, there is no difference between *unordered* and *not atomic* for objects of size less or equal to 64 bits. *Not atomic* instructions also permit a large variety of optimizations. However, this is not a problem as DIVINE should be applied on the bitcode after any desired optimizations.

organized in a FIFO manner; it is flushed nondeterministically in any order which satisfies the condition that no entry can be flushed into the memory if there is an older *matching entry*. Entry A matches entry B (or depends on B) if both A and B change the same memory location (this does not imply that the address in A is the same as the address in B, as it can happen that B changes only a part of the value written by A or vice versa).

Furthermore, the entry can be set as flushed using the flushed flag or deleted from the store buffer when it is flushed. The flushed flag is used only for *monotonic* (or stronger) entries which follow any *release* (or stronger) entries; all other entries are deleted immediately. These flushed entries are used to check validity of the run.

The description of the realization of atomic instructions follows. We will denote *local store buffer* to be the store buffer of the thread which performs the instruction in question; the store buffers of all other threads will be denoted as *foreign store buffers*.

All stores are performed into the local store buffer. The address, the value, and the bitwidth of the value is saved, the atomic ordering of the entry is set according to the atomic ordering of the corresponding `store` instruction, the flushed flag is set to false and the observed set is set to the empty set.

***Unordered* loads** can be executed at any time. All loads load the value from the local store buffer if it contains a newer value then the memory.

***Monotonic* load** can be executed at any time too. Furthermore, if there is a flushed, at least *monotonic* entry E in any foreign store buffer, the observed flag is set to any entry which:

- is in the same store buffer as E and is older, or E itself,
- and it has at least a *release* ordering.

All these entries are set to be observed by the thread which performs the load.

***Monotonic* atomic compound instruction** (`cmpxchg` or `atomicrmw`) can be performed if a *monotonic* load can be performed and there is no not-flushed *monotonic* entry for the same memory location in any foreign store buffer. It also sets observed flags in the same way as *monotonic* loads do.

***Acquire* fence** can be performed if there are no entries in foreign store buffers with at least *release* ordering which were observed by the current thread. This way, a *release* store or fence synchronizes with an *acquire* fence if the conditions of fence synchronization are met.

***Acquire* load** can be performed if

- a *monotonic* load from the same memory location can be performed,
- and an *acquire* fence can be performed,
- and there are no flushed *release* (or stronger) store entries for the same memory location in any foreign store buffer.

This way an *acquire* load synchronizes with the latest *release* store to the same memory location if the value of the store can be already read (the only way to remove a *release* entry from a store buffer is to first remove all the entries which precede it).

***Acquire* atomic compound operations** can be performed if

- an *acquire* load from the same memory location can be performed,
- and there are no (at least) *release* entries for the same memory location in any foreign store buffer.

***Release* and *acquire-release* loads** are not allowed by LLVM.

***Release* fences** add a fence entry into the local store buffer. The memory ordering of the entry is set according to the ordering of the fence and the observed set is set to an empty set.

***Acquire-release* fence** behaves as both a *release* and an *acquire* fence.

***Sequentially consistent* fence** can be performed if an *acquire* fence can be performed and there are no *sequentially consistent* entries in any foreign store buffer. This way, a *sequentially consistent* fence synchronizes with any *sequentially consistent* operations performed earlier.

***Sequentially consistent* load or atomic compound operation** can be performed if

- the same operation with an *acquire-release* ordering and on the same memory location can be performed,
- and a *sequentially consistent* fence can be performed.

While there is no explicit synchronization between multiple *sequentially consistent* stores/loads/fences there is still a total order of all the *sequentially consistent* operations which respects the program order of each of the threads and the synchronizes-with edges. For operations within a single thread, their relative position in this total order is given by the order in which they are executed. For two stores executed in different threads which are not ordered as a result of an explicit synchronization, their relative order can be arbitrary as they are not dependent. Loads and atomic compound operations are explicitly synchronized as described above.

The case of *monotonic* operations is similar, not-otherwise-synchronized stores and loads from different threads can be flushed in an arbitrary order. The total order of *monotonic* operations over a memory location can be derived from their order of execution:

- the total order of `store` instructions is given by the order in which the corresponding store entries are flushed (which is a total order as DIVINE executes instructions interleaved and not in parallel);
- the total order of `load` instructions is given by the order they are executed in;
- every `store` is ordered before any `load` which loads the value written by this or any later stores;
- this total order is consistent with the order of execution of threads; for `load` instructions this is obvious, for `store` it follows from the fact that stores to the same memory location from the same thread cannot be reordered.

In the case of `atomicrmw` and `cmpxchg` instructions, stronger synchronization is needed; representing them as an atomically-executed `load` followed by a `store` could break the total order. Suppose thread 0 performs an atomic increment of a memory location `@x` and later thread 1 increments the same location; now, if the store buffer entry corresponding to the `store` in thread 0 is not flushed before the `load` in thread 1, the old value will be read in thread 1 and the result will be the same as if only one increment executed. The corresponding ordering is: `load` in thread 0, `load` in thread 1, `store` in thread 0, and `store` in thread 1. This ordering is possible even though both of the `load`–`store` combinations are executed atomically, due to the fact that the position of a `store` in the total order is determined by the moment in which this store is flushed. To resolve this, these atomic operations can only be performed if there are no atomic store entries for the given memory location in any foreign store buffer. This way, a total ordering of these operations is guaranteed.

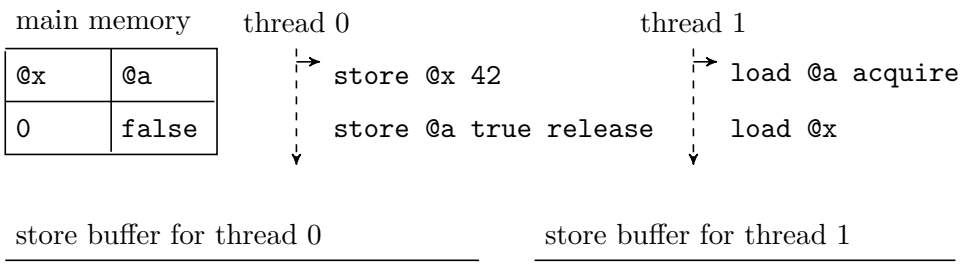
Figures 4.8, 4.9, and 4.10 demonstrate the store buffer approximation of the LLVM memory model for the case of simple shared variables, one of which is accessed atomically. Figures 4.11, 4.12, and 4.13 show an illustration with a `fence` instruction.

4.4.2 Nondeterministic Flushing

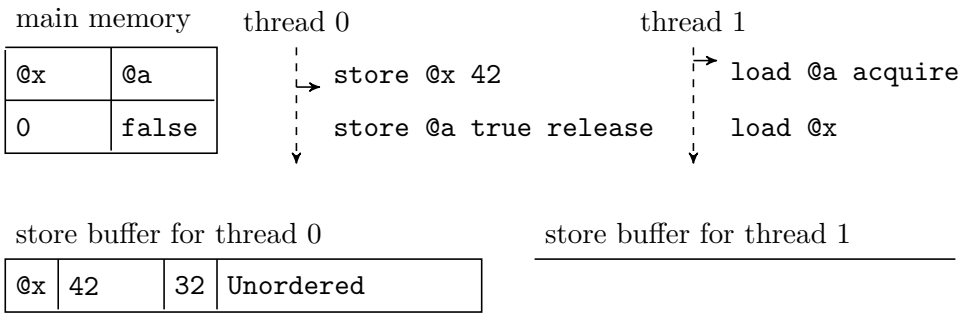
When a write is performed into a store buffer, it can be flushed into memory at any later time. To simulate this nondeterminism, we introduce a thread which is responsible for store buffer flushing. There will be one such *flusher* thread for each store buffer. The interleaving of this thread with the other threads will result in all the possible ways in which flushing can be done.

```
int x;
std::atomic< bool > a;
void thread0() {
    x = 42;
    a.store( true, std::memory_order_release );
}
void thread1() {
    while ( !a.load( std::memory_order_acquire ) { }
    std::cout << x << std::endl; // always prints 42
}
```

This is an example of two threads which communicate using a shared global variable `x` which is guarded by an atomic global variable `a`. Following is a simplified execution of this programs (only `load` and `store` instructions are shown).

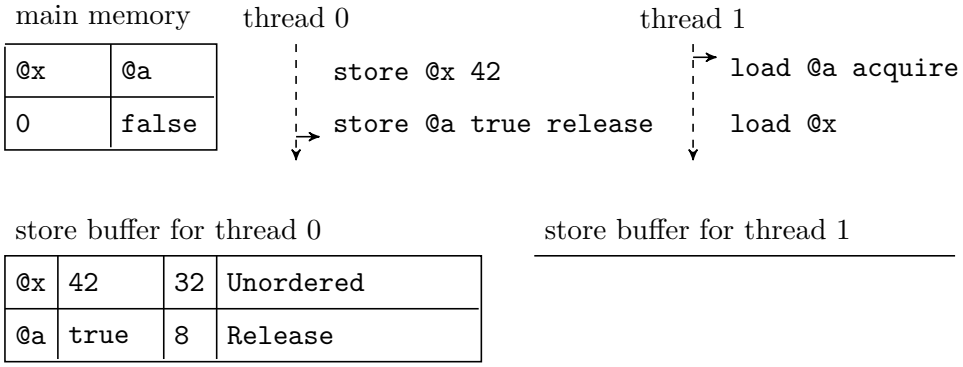


1. Before the first instruction is executed, `@x` is initiated to 0 and `@a` to `false`. Store buffers are empty. When thread 0 executes the first instruction, the store will be performed into store buffer.

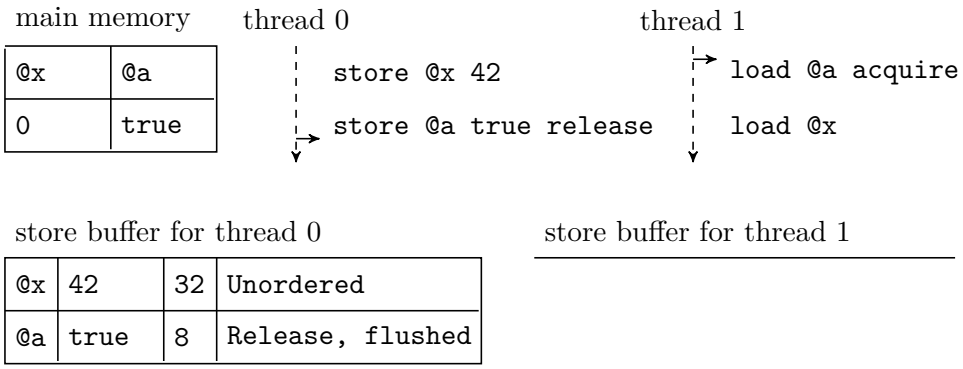


2. After the first instruction of thread 0, its store buffer contains an entry with the address of the stored memory location, the stored value, its bitwidth, and the memory ordering used for the store.

Figure 4.8: Example of the weak memory model simulation with store buffers, part I.

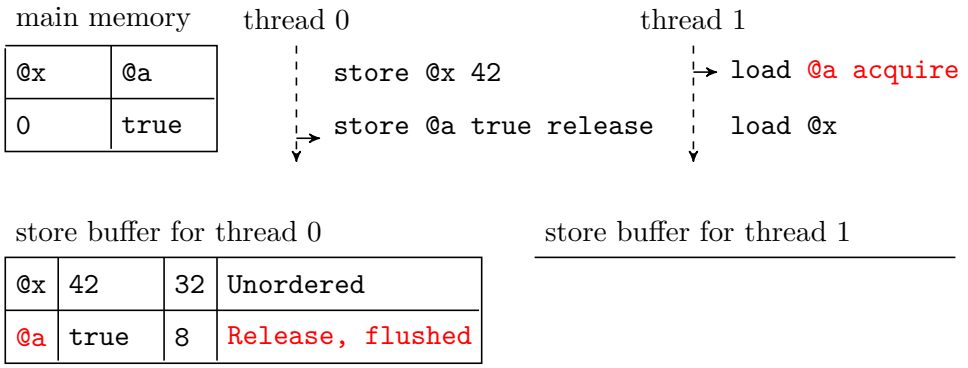


3. Second entry is appended to the store buffer. If the first instruction of thread 1 executed now, it would read `false` from the memory and the cycle would be repeated.

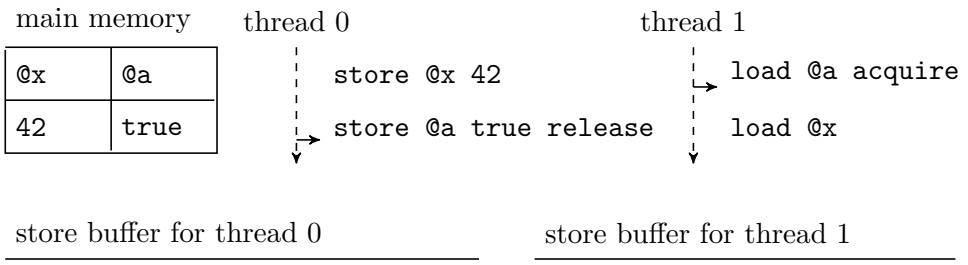


4. The entry for `@a` in the store buffer of thread 0 is flushed into memory, but the entry is still remembered in the store buffer as it is a *release* entry and future loads (if they have at least an *acquire* ordering) will have to synchronize with it. It would be also possible to first flush the entry for `@x`; in this case it would be removed from the store buffer as it is the oldest entry, and therefore no explicit synchronization is necessary.

Figure 4.9: Example of the weak memory model simulation with store buffers, part II.



5. When the first instruction of thread 1 is executed, a synchronization takes place. The *acquire* load on `@a` forces the matching, flushed entry in the store buffer of thread 0 to be evicted; however, this is a *release* entry so all the entries which precede it will have to be flushed and evicted too.

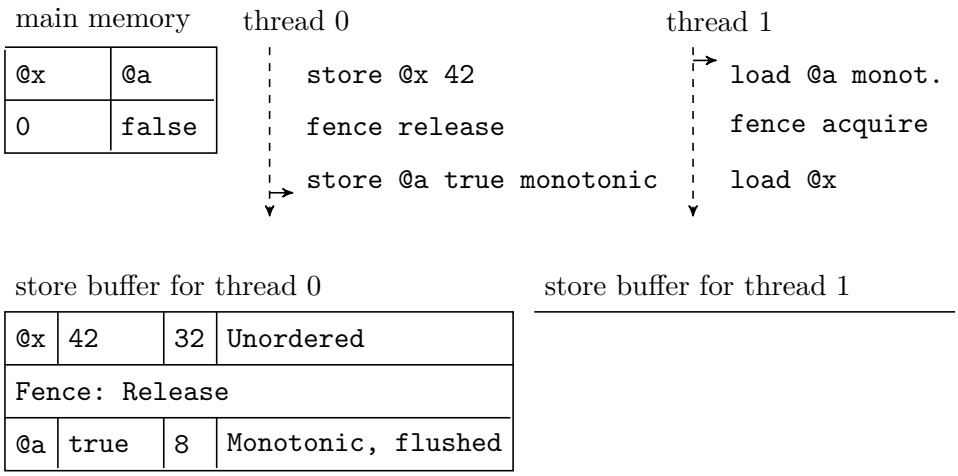


6. The load of `@a` in thread 1 now proceeds and the load of `@x` will always return `42` as there is a synchronizes-with edge between the *release* store and the *acquire* load of `@a` and therefore all action of thread 0 before the store of `@a` are visible after the load of `@a` returns the stored value.

Figure 4.10: Example of the weak memory model simulation with store buffers, part III.

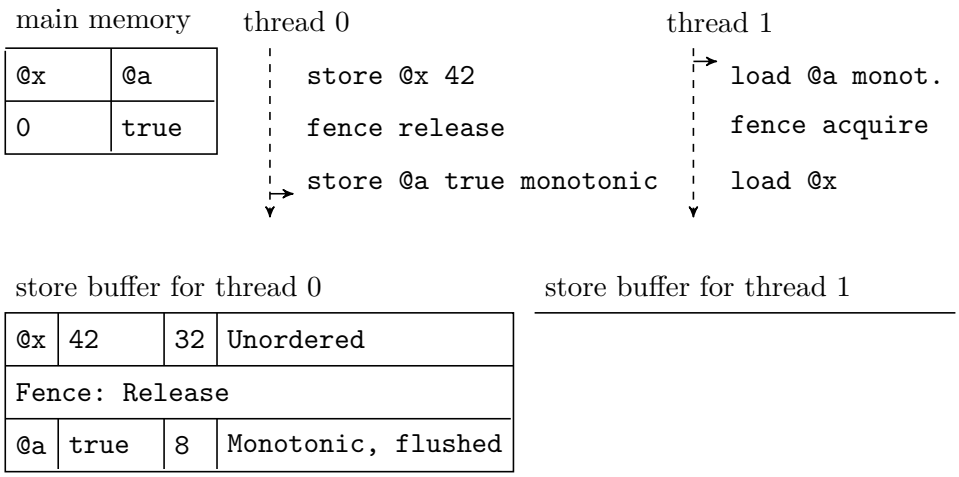
```
1  int x;
2  std::atomic< true > a;
3
4  void thread0() {
5      x = 42;
6      std::atomic_thread_fence( std::memory_order_release );
7      a.store( true, std::memory_order_monotonic );
8  }
9
10 void thread1() {
11     while ( !a.load( std::memory_order_relaxed ) { }
12     std::cout << x << std::endl; // can print 0 or 42
13     std::atomic_thread_fence( std::memory_order_acquire );
14     std::cout << x << std::endl; // always prints 42
15 }
```

This example is similar to the one in Figure 4.8; however, it uses explicit fences to synchronize the access to the global variable `x`.

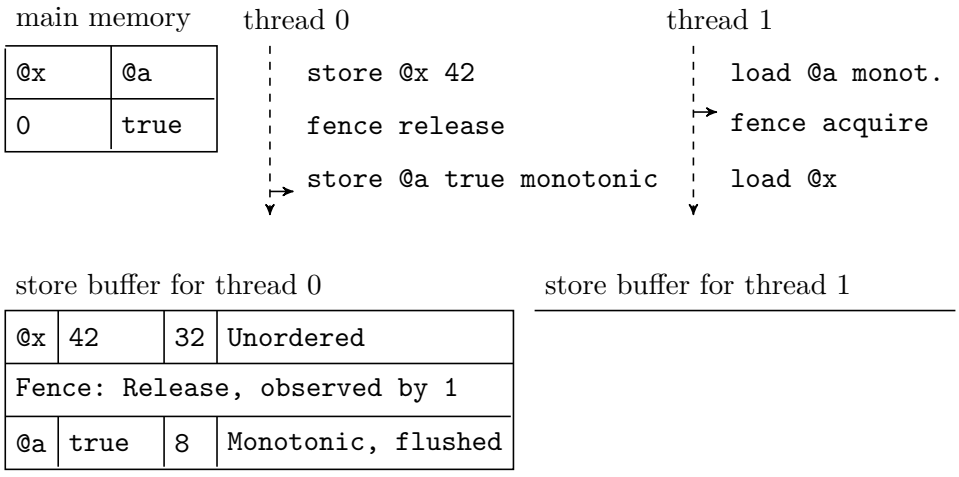


1. After all the instructions of thread 0 executed, the store buffer contains two store entries and one fence entry which corresponds to the fence on line 6.

Figure 4.11: Example of the weak memory model simulation with fences, part I.

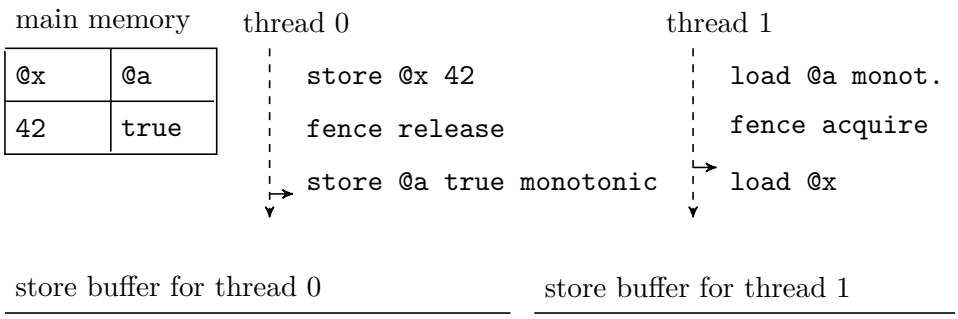


2. The last entry from the store buffer is flushed but the entry remains in the store buffer as it is preceded by a *release* entry.



3. The *monotonic* load of `@a` executes; the value is already flushed into memory and the load does not cause any synchronization. It does, however, add an *observed* flag with thread ID of the thread which performed the load to any at least *release* barrier which precedes the store buffer entry for `@a`. The observed flag would be also added to any *release* (or stronger) store entries which precede the store entry for `@a` and to the entry for `@a` if it was *release* or stronger.

Figure 4.12: Example of the weak memory model simulation with fences, part II.



4. The fence executes. It is an *acquire* fence so it synchronizes with any (at least) *release* fence which was observed by the thread which executed the *release* fence (thread 1). This means that all the entries before all the observed *release* fences have to be flushed and evicted and the fence is flushed and evicted too. Finally, as the store entry for `@a` was already flushed and it would be the first entry in the store buffer after the fence is evicted, it is also evicted. The load of `@x` will always return `42`.

Figure 4.13: Example of the weak memory model simulation with fences, part III.

The flusher threads run an infinite loop; an iteration of this loop is enabled if there are any entries in the store buffer associated with this flusher thread. In each iteration of the loop, the flusher thread nondeterministically selects an entry in the store buffer and flushes if it is possible (if there is no older entry for a matching location).

4.4.3 Atomic Instruction Representation

Atomic instructions (`cmpxchg` and `atomicrmw`) are not transformed to the LLVM memory model directly. Instead, they are first split into a sequence of instructions which performs the same action (but not atomically) and this sequence is executed under DIVINE's mask. This sequence of instructions contains loads and stores with atomic ordering derived from the atomic ordering of the original atomic instruction and these instructions are later transformed to the LLVM memory model. The sequence also contains an explicit additional synchronization required to ensure a total ordering of all the atomic instructions over the same memory location.

```
%res = atomicrmw op ty* %pointer, ty %value ordering
```

The atomic read-modify-write instruction atomically performs a load from `pointer` with the given atomic ordering, then it performs a given operation with the result of the load and `value`, and finally it stores the result into the

`pointer` again, using the given atomic ordering. It yields the original value loaded from `pointer`. The operation `op` can be one of `exchange`, `add`, `sub`, `and`, `or`, `nand`, `xor`, `max`, `min`, `umax`, `umin` (the last two are unsigned minimum and maximum, while the previous two perform signed versions). An example of the transformation of this instruction can be seen in Figure 4.14.

```
%res = cmpxchg ty* %pointer, ty %cmp, ty %new
    success_ordering failure_ordering ; yields  { ty, i1 }
```

The atomic compare-and-exchange instruction atomically loads a value from `pointer`, compares it with `cmp` and, if they match, stores `new` into `pointer`. It returns a tuple which contains the original value loaded from `pointer` and a boolean flag which indicates if the comparison succeeded. Unlike the other atomic instructions, `cmpxchg` takes two atomic ordering arguments; one which gives the ordering in the case of success and the other for ordering in the case of failure. This instruction can be replaced by code which performs a `load` with `failure_ordering`, comparison of the loaded value and `cmp` and, if the comparison succeeds, an additional synchronization with `succeeds_ordering` and a `store` with `succeeds_ordering`. The reason to use `failure_ordering` in the load is that a failed `cmpxchg` should be equivalent to a load with `failure_ordering`. The additional synchronization in the case of success is needed to strengthen the ordering to `success_ordering` and to ensure a total store order of all operations which affect the given memory location. An example of such a transformation can be seen in Figure 4.15.

4.4.4 Memory Order Specification

It is not always desirable to verify a program with the weakest possible memory model. For this reason, the transformation can be parametrized with a minimal ordering it guarantees for a given memory operation (each of `load`, `store`, `fence`, `atomicrmw`, `cmpxchg` success ordering, and `cmpxchg` failure ordering can be specified).

This way, memory models stronger than the LLVM memory model can be simulated, for example total store order is equivalent to setting all of the minimal orderings to *release-acquire*, the memory model of `x86` (which is basically TSO with *sequentially consistent* atomic compare and swap, atomic read-modify-write, and fence) can be approximated by setting `load` to *acquire*, `store` to *release*, and the remaining instructions to *sequentially consistent*.

4.4.5 Memory Cleanup

When a write to a certain memory location is delayed, it can happen that this memory location becomes invalid before the delayed write is actually performed. This can happen both for local variables and for dynamically allocated memory. For local variables, the value might be written after the

```
; some instructions before
%res = atomicrmw op ty* %pointer, ty %value ordering
; some instructions after
```

This will be transformed into:

```
; some instructions before
%0 = call i32 @__divine_interrupt_mask()
%atomicrmw.shouldunlock = icmp eq i32 %3, 0
%atomicrmw.orig = load atomic ty, ty* %ptr ordering
; explicit synchronization
%1 = bitcast ty * %ptr to i8*
call void @__lart_weakmem_sync(i8* %1, i32 width, i32 ordering)
; the instruction used here depends on op:
%opval = op %atomicrmw.orig %value
store atomic ty %opval, ty* %ptr seq_cst
br i1 %atomicrmw.shouldunlock,
    label %atomicrmw.unmask,
    label %atomicrmw.continue

atomicrmw.unmask:
call void @__divine_interrupt_unmask()
br label %atomicrmw.continue

atomicrmw.continue:
; some other instructions after, %res is replaced with
; %atomicrmw.orig
```

The implementation of `op` depends on its value, for example for `exchange` there will be no instruction corresponding to `op` and the `store` will store `%value` instead of `%opval`. On the other hand, `max` will be implemented using two instructions (first the values are compared, then the bigger of them is selected using the `select` instruction):

```
%1 = icmp sgt %atomicrmw.orig %value
%opval = select %1 %atomicrmw.orig %value
```

Figure 4.14: An example of the transformation of `atomicrmw` instruction into an equivalent sequence of instructions which is executed atomically using `__divine_interrupt_mask` and synchronized strongly with the other operations using `__lart_weakmem_sync`.

```
; some instructions before
%res = cmpxchg ty* %pointer, ty %cmp, ty %new
    success_ordering failure_ordering
; some instructions after
```

This will be transformed into:

```
; some instructions before
%0 = call i32 @__divine_interrupt_mask()
%cmpxchg.shouldunlock = icmp eq i32 %6, 0
%cmpxchg.orig = load atomic ty, ty* %ptr failure_ordering
%1 = bitcast ty * %ptr to i8*
call void @__lart_weakmem_sync(i8* %1, i32 bitwidth,
                               i32 failure_ordering)
%cmpxchg.eq = icmp eq i64 %cmpxchg.orig, %cmp
br i1 %cmpxchg.eq,
    label %cmpxchg.ifeq,
    label %cmpxchg.end

cmpxchg.ifeq:
call void @__lart_weakmem_sync(i8* %1, i32 bitwidth,
                               i32 success_ordering)
store atomic ty %new, ty* %ptr success_ordering
br label %cmpxchg.end

cmpxchg.end:
%2 = insertvalue { ty, i1 } undef, ty %cmpxchg.orig, 0
%res = insertvalue { ty, i1 } %2, i1 %cmpxchg.eq, 1
br i1 %cmpxchg.shouldunlock,
    label %cmpxchg.unmask,
    label %cmpxchg.continue

cmpxchg.unmask:
call void @__divine_interrupt_unmask()
br label %cmpxchg.continue

cmpxchg.continue:
; some instructions after
```

Figure 4.15: An example of the transformation of the `cmpxchg` instruction into an equivalent sequence of instructions which is executed atomically using `__divine_interrupt_mask` and synchronized strongly with the other operations using `__lart_weakmem_sync`.

function exits, while for dynamic memory, the value might be stored after the memory is freed.

To solve this problem, entries corresponding to invalidated memory need to be removed from the local store buffer. The reason to leave the entries in foreign store buffers is that the existence of such entries suggests that the write to the (soon-to-be invalidated) memory location did not synchronize properly with the end of the scope of the memory location.

For dynamic memory, it is sufficient to remove all entries corresponding to the object just before the call to `__divine_free` which performs the deallocation. For local variables, it is necessary to remove the entries just before the function exits and to do this we employ the local variable cleanups described in Section 4.2.3.

4.4.6 Integration with $\tau+$ Reduction

As described in Section 3.2.5, one of the important reduction techniques in DIVINE is the $\tau+$ reduction, which allows execution of multiple consecutive instructions in one atomic block if there is no more then one action observable by other threads in this block. For example, a `store` instruction is observable if and only if it stores to a memory block to which some other thread holds a pointer.

This means that any load from or any store into store buffer will be considered to be a visible action, because the store buffer has to be visible both from the thread executing the load or the store and from the thread which flushes the store buffer to memory.

To partially mitigate this issue, it was proposed in [42] to bypass the store buffer when storing to addresses which are considered thread private by DIVINE's $\tau+$ reduction. To do this, the `__divine_is_private` intrinsic function is used in the function which implements weak memory store, and, if the address to which the store is performed is indeed private, the store is executed directly, bypassing the store buffer.

This reduction is indeed correct for TSO stores which were simulated in [42]. It is easy to see that the reduction is correct if a memory location is always private or always public for the entire run of the program. The first case means that it is never accessed from more then one thread and therefore no store buffer is needed, the second case means the store buffer will be used always. If the memory location (say `x`) becomes public during the run of the program, it is again correct (the publication can happen only by writing an address of memory location from which `x` can be reached following pointers into some already-public location):

- if `x` is first written and later published, then, were the store buffers used, the value of `x` would need to be flushed from the store buffer before `x`

could be reached from the other thread (because the stores cannot be reordered under TSO), and therefore the observed values are the same with and without the reduction;

- if `x` is first made private and then written, then the "making private" must happen by changing some pointer in a public location, an action which will be delayed by the store buffer. However, this action must be flushed before the store to `x` in which it is considered private, as otherwise `x` would not be private, and therefore also before any other modifications to `x` which precede making `x` private;

- the remaining possibilities (`x` written after the publication, and `x` written before making it private) are not changed by the reduction.

In the case of the general LLVM memory model with presence of explicit atomic instructions, this reduction cannot be used: suppose a memory location `x` is written while it is thread-private and later the address of `x` is written into a visible location `y`. Now it might be possible, provided that `y` has weaker than *release* ordering that the old value of `x` is accessed through `y` (if `y` is flushed before `x`). For this reason, stores to all memory locations have to go through the store buffer (unless it is possible to prove that they can never be visible by more than one thread).

Furthermore, considering that all store buffers are reachable from all threads, and therefore any memory location which has an entry in the store buffer is considered public, we can bypass store buffers for `load` instructions, even under LLVM memory model. That is, the store buffer lookup can be bypassed for a memory location if it is considered private by DIVINE, because no memory location which is private can have an entry in any store buffer. This means that loads of private memory locations are no longer considered as visible actions by $\tau+$, which leads to a state space reduction.

As a final optimization, any load from or store into a local variable which never escapes the scope of the function which allocated it need not be instrumented; that is, `load` or `store` instruction need not be replaced with the appropriate weak-memory-simulating version. To detect these cases, we currently use LLVM's `PointerMayBeCaptured` function to check if the memory location of a local variable was ever written to some other memory location. A more precise approximation could use pointer analysis to detect which memory locations can only be accessed from one thread.

The evaluation of the proposed optimizations can be found in Section 5.2.1.

4.4.7 Interaction With Atomic Sections

An important consequence of the LLVM memory model transformation is that effects of instructions which are executed inside DIVINE's atomic sections

(using `__divine_interrupt_mask`) need not happen as part of the atomic section. For example, a `store` executed in an atomic section can be flushed much later after the atomic sections ends. This creates additional requirements to the implementation of libraries for DIVINE, namely the `pthread` threading library. For this reason, any `pthread` function which uses atomic sections now includes a *sequentially consistent* fence after the atomic section is entered and before it is exited.

4.4.8 Implementation

The transformation implementation consists of two passes over LLVM bitcode, the first one is used to split loads and stores larger than 64 bits into smaller loads and stores. In the second phase, the bitcode is instrumented with the LLVM memory model using functions which implement stores, loads, and fences using store buffers and atomic functions are rewritten as described in Section 4.4.3. The userspace functions simulating the memory model are implemented in C++ and compiled together with the verified program by `divine compile`. The userspace functions can be found in `lart/userspace/weakmem.h` and `lart/userspace/weakmem.cpp`, the transformation pass can be found in `lart/weakmem/pass.cpp`. The transformation can be run using the `lart` binary, see Section A.2.4 for details on how to compile and run LART.

Userspace Functions

The userspace interface is described by `lart/userspace/weakmem.h`, which defines types and functions necessary for the transformation.

```
volatile extern int __lart_weakmem_buffer_size;
```

The store buffer size limit is saved in this variable so that it can be set by the weak memory transformation.

```
enum __lart_weakmem_order {
    __lart_weakmem_order_unordered,
    __lart_weakmem_order_monotonic,
    __lart_weakmem_order_acquire,
    __lart_weakmem_order_release,
    __lart_weakmem_order_acq_rel,
    __lart_weakmem_order_seq_cst
};
```

An enumeration type which corresponds to LLVM atomic orderings.

```
void __lart_weakmem_store( char *addr, uint64_t value,
            uint32_t bitwidth, __lart_weakmem_order ord );
```

```
uint64_t __lart_weakmem_load( char *addr, uint32_t bitwidth,
            __lart_weakmem_order ord );
void __lart_weakmem_fence( __lart_weakmem_order ord );
```

These functions replace `store`, `load`, and `fence` instructions. The transformation is expected to fill in the `bitwidth` parameter according to the actual bit width of the loaded/stored type and to perform any necessary casts. Each of these functions is performed atomically using DIVINE's atomic mask. The load function must be able to reconstruct the loaded value from several entries in the store buffer as it is possible that some entry corresponds to only a part of the requested value. While these functions are primarily intended to be used by the LART transformation, their careful manual usage can be used to manually simulate a weak memory model for a subset of operations.

```
void __lart_weakmem_sync( char *addr, uint32_t bitwidth,
                          __lart_weakmem_order ord );
```

This function is used for explicit synchronization of atomic instructions (`atomicrmw`, `cmpxchg`) to ensure a total order of all atomic modifications. The memory order must be at least *monotonic*, this function ensures that there is no (at least) *monotonic* entry for a matching address in any foreign store buffer.

```
void __lart_weakmem_cleanup( int cnt, ... );
```

This function is used to implement memory cleanups (Section 4.4.5). Its variadic arguments are memory addresses which should be evicted from the local store buffer, `cnt` should be set to the number of addresses provided.

```
void __lart_weakmem_memmove( char *dest, const char *src,
                                         size_t n );
void __lart_weakmem_memcpy( char *dest, const char *src,
                                         size_t n );
void __lart_weakmem_memset( char *dest, int c, size_t n );
```

These functions are used as replacements for `llvm.memcpy`, `llvm.memset`, and `llvm.memmove` intrinsics. The transformation pass will derive two versions of these functions, one to be used by the weak memory model implementation (this version must not use store buffers) and the other to implement these intrinsics in the weak memory model.

All of these functions have following attributes (using GCC/Clang syntax `__attribute__`):

`noinline` to prevent inlining of these functions into their callers;

`flatten` to inline all function calls these functions contain into their body (this is used to make sure these functions do not call any function which could use store buffers);

`annotate("lart.weakmem.bypass")` which indicates to the transformation pass that these functions should not be transformed to use store buffers;

`annotate("lart.weakmem.propagate")` which indicates to the transformation pass that any function called from these functions should not be transformed to use store buffers (this is done to handle cases in which the compiler refuses to inline all calls into these functions; the transformation pass will output a warning if this happens).

LART Transformation Pass

The transformation pass processes all the functions in the module one by one. For the weak memory implementation functions, it only transforms any calls to `llvm.memmove`, `llvm.memcpy`, and `llvm.memset` intrinsics to calls to their implementations which do not use store buffer simulation.

For other functions, the transformation is done in the following three phases.

1. Atomic compound instructions (`atomicrmw` and `cmpxchg`) are replaced by their equivalents as described in Section 4.4.3.

2. Loads and stores are replaced by calls to the appropriate userspace functions. This includes casting of the addresses to the `i8*` LLVM type and values need to be truncated or respectively extended to the `i64` type. For non-integral types, this also includes a `bitcast` from (to) integral types (a cast which does not change the bit pattern of the value, changing only its type).

   The atomic ordering used in the LLVM memory model simulation is derived from the atomic ordering of the original instruction and from the default atomic ordering for a given instruction type which is determined by the configuration of the transformation (Section 4.4.4).

3. Memory intrinsics (`llvm.memmove`, `llvm.memcpy`, and `llvm.memset`) are translated to the appropriate userspace functions which implement these operations using store buffers.

The transformation is not applied to instructions which manipulate local variables which do not escape the scope of the function which defines them.

The transformation is configurable. It can be specified what minimal atomic ordering is guaranteed for each instruction type and what should be the size bound for store buffers. The specification is given when LART is invoked (see Section A.2.4), for atomic orderings it can be one of:

`std` for the unconstrained LLVM memory model;

`tso` for Total Store Order simulation; this guarantees that all loads have at least an *acquire* ordering, all stores have at least a *release* ordering, and all the other operations have at least an *acquire-release* ordering.

`x86` for simulation of a memory model similar to the one in `x86` CPUs. In this case, loads have at least an *acquire* ordering, stores have at least a *release* ordering and all other transformed operations have a *sequentially consistent* ordering.

custom specification is a comma separated list of `kind=ordering` pairs, where `kind` is an instruction type (one of `all`, `load`, `store`, `cas`, `casfail`, `casok`, `armw`, and `fence`) and `ordering` is the atomic ordering specification (one of `unordered`, `relaxed`,[5] `acquire`, `release`, `acq_rel`, and `seq_cst`). The list of these pairs is processed left to right so that later entries override earlier ones.

For example, TSO can be specified as `all=acq_rel`, the equivalent of `x86` can be specified as `all=seq_cst,load=acquire,store=release`.

4.5 Code Optimization in Formal Verification

DIVINE aims at verification of real-world code written in C and C++. Both LLVM IR and assembly produced from such code is often heavily optimized during the compilation to increase its execution speed. To verify the code as precisely as possible, it is desirable to verify the LLVM IR with all the optimizations which will be used in the binary version of the program and the binary should be compiled by the same compiler as the LLVM IR used in DIVINE. Ideally, it would be possible to use the same LLVM IR for verification and to build the binary. This is, however, not currently possible as DIVINE needs to re-implement library features (namely `pthreads` and C++ exception handling) and this implementation might not be compatible with the implementation used on given platform. Nevertheless, DIVINE should use the optimization levels requested by its user for the program compilation.

On the other hand, it is desirable to utilize LLVM optimizations in such a way that model checking can benefit from it. This, however, requires special purpose optimizations designed for the verification, as the general purpose optimizations do not meet two critical requirements for verification.

- **They can change satisfiability of the verified property.** This is usually caused by the fact that compiler optimizations are not required to preserve behaviour of parallel programs, and that many programs

[5] In this case `relaxed` is used to denote the LLVM's *monotonic* ordering to match the name used for this ordering in the C++11/C11 standard.

written in C/C++ contain undefined behaviour as they access non-atomic non-volatile variables from multiple threads. See Figure 4.16 for an example of such a property-changing optimization.

- **They might increase state space size.** Not all optimizations which lead to faster execution lead to faster verification as they might change program behaviour in such a way that the model checker generates more states. An example of such a transformation can be any transformation which increases the number of registers in a function. This might cause states which were originally considered to be the same to become distinct after the optimization. More specifically, examples of such transformation are promotion of variables into registers, loop unrolling, and loop rotation which can be seen in Figure 4.17.

For these reasons, we suggest some optimization techniques which would allow optimization of LLVM IR but not change the verification outcome or increase the state space size. On the other hand, these techniques can use specific knowledge about the verification environment they will be used in. Some of these techniques were already implemented as part of this thesis and are evaluated in Section 5.3, some of them are proposals for future work.

4.5.1 Constant Local Variable Elimination

Especially with optimizations disabled, compilers often create `alloca` instructions (which correspond to stack-allocated local variables) even for local variables which need not have an address and perform loads and stores into those memory locations instead of keeping the value in registers. To eliminate unnecessary `alloca` instructions, LLVM provides a register promotion pass. Nevertheless, this pass is not well suited for model checking as it can add registers into the function and in this way increase the state space size. For this reason, we introduce a pass which eliminates *constant* local variables, as these can be eliminated without adding registers (in fact, some registers can be removed in this case).

With this reduction, an `alloca` instruction can be eliminated if the following conditions are met:

- the address of the memory is never accessed outside of the function;
- it is written only once;
- the `store` into the `alloca` dominates all `loads` from it.

The first condition ensures that the `alloca` can be deleted, while the other two conditions ensure that the value which is loaded from it is always the same, and therefore can be replaced with the value which was stored into the `alloca`.

```
int x;
void foo() {
    x = 1;
    assert( x == 1 );
}
int main() {
    std::thread t( &foo );
    x = 2;
    t.join();
}
```

This code is an example of an undefined behaviour. The global non-atomic variable `x` is written concurrently from two threads. For this program, assertion safety does not hold: the assertion can be violated if the assignment `x = 2` executes between `x = 1` and `assert( x == 1 )`.

```
store i32 1, i32* @x, align 4
%0 = load i32, i32* @x, align 4
%tobool = icmp ne i32 %0, 0
%conv = zext i1 %tobool to i32
call void @__divine_assert(i32 %conv)
ret void
```

The body of `foo` emitted by Clang without any optimization is a straightforward translation of the C++ code. It stores into global `@x`, then loads it and compares the loaded value to `0`. In this case, DIVINE will report an assertion violation.

```
store i32 1, i32* @x, align 4
tail call void @__divine_assert(i32 1)
ret void
```

This is optimized (`-O2`) version of `foo`. The `store` is still present, but the compiler assumes that the `load` which should follow it will return the save value as written immediately before it (this is a valid assumption for non-atomic, non-volatile shared variable). For this reason, the assertion is optimized into `assert( true )` and no assertion violation is possible.

Figure 4.16: An example of program in which optimizations change whether a property holds.

Suppose a program with a global atomic boolean variable `turn` and a code snippet which waits for this value to be set to true:

```
while ( !turn ) { }
// rest of the code
```

This program might generate following LLVM:

```
loop:
%0 = load atomic i8, i8* @turn seq_cst
%1 = icmp eq i8 %0, 0
br %1, label %loop, label %end

end:
; rest of the code
```

With optimization, this LLVM can be changed to:

```
pre:
%0 = load atomic i8, i8* @turn seq_cst
%1 = icmp eq i8 %0, 0
br %1, label %loop, label %end

loop:
%2 = load atomic i8, i8* @turn seq_cst
%3 = icmp eq i8 %2, 0
br %3, label %loop, label %end

end:
; rest of the code
```

Basically the loop is rotated to a loop equivalent to the following code:

```
if ( !trun ) {
    do { } while ( !turn ) { }
}
```

While this code might be faster in practice due to branch prediction, for model checking this is an adverse change as the model checker can now distinguish the state after one and two executions of the original loop based on register values.

Figure 4.17: An example of optimization with an adverse effect on model checking.

In the current implementation of this pass, each function is searched for `alloca` instructions which meet these criteria (ignoring uses of address in the `llvm.dbg.declare` intrinsic[6]) and all uses of results of loads from these memory locations are replaced with the value which was originally stored into it; finally the `alloca` and all its uses are eliminated from the function. Please note that the conditions ensure that the only uses of the `alloca` are the single store into it, the loads which read it, and `llvm.dbg.declare` intrinsics.

4.5.2 Constant Global Variable Annotation

In DIVINE, any non-constant global variable is considered to be visible by all threads and is saved in each state in the state space. However, it can happen that this variable cannot be changed during any run of the program. If such a condition can be detected statically, it is possible to set this variable to be constant which removes it from all states (it is stored in constants, which are part of the interpreter state) and it also causes the loads of this variable to be considered to be invisible actions by $\tau+$ reduction.

For a global variable to be made constant in this way, it must meet the following conditions:

- it must be never written to, neither directly nor through any pointer;
- it must have a constant initializer.

While the second condition can be checked from the definition of the global variable, the first one cannot be exactly efficiently determined. It can be approximated using pointer analysis.

Currently, LART lacks working pointer analysis, so we used a simple heuristic for the initial implementation: the address of the global variable must not be stored into any memory location and any value derived from the address must not be used in instructions which can store into it (`store`, `atomicrmw`, `cmpxchg`). This is implemented by recursively tracking all uses of the values derived from the global variable's address. The implementation is available in `lart/reduction/globals.cpp`.

4.5.3 Local Variable and Register Zeroing

LLVM registers are immutable and therefore they retain their value even after it is no longer useful. This means that there can be states in the state space which differ only in a value of a register which does not change the execution of the program as it will be never used again. This situation can be eliminated by setting the no-longer-used registers to 0. However, this is not possible in LLVM as it is in a static single assignment form.

[6] This intrinsic is used to bind debugging information such as a variable name with the variable's LLVM IR representation. It does not affect the behaviour of the program in any way.

Nevertheless, with addition of one intrinsic function into the DIVINE's LLVM interpreter, it is possible to zero registers in DIVINE at the place determined by the call to this intrinsic. Since LLVM is type safe, this intrinsic is actually implemented as a family of functions with `__divine_drop_register.` prefix, one for each type of register which needs to be zeroed. Signatures for these functions are generated automatically by the LART pass which performs register zeroing. Any call to a function with this prefix is implemented as an intrinsic which zeroes the register and sets it as uninitialized.

The LART pass (which is implemented in `lart/reduce/register.cpp`) processes each function with the following algorithm.

1. For each instruction i, it searches for last uses:
   - this is either a use u such that no other use of i is reachable from u;
   - or a use u which is part of a loop and all the uses of i reachable from it are in the same loop.
2. Insertion points for `__divine_drop_register` calls are determined:
   - for the uses which are not in a loop, the insertion point is set to be immediately after the use;
   - for the uses which are in a loop, the insertion point is at the beginning of any basic block which follows the loop.

   Strongly connected components of the control flow graph of the function are used to determine if an instruction is in a loop and successors of a loop.
3. If an instruction i dominates an insertion point, i is dropped at this point using `__divine_drop_register`.

Furthermore, if the instruction in question is an `alloca`, it is treated specially. `alloca` cannot be zeroed until the local variable it represents is released. A simple heuristic is used to determine if the local variable might be aliased, and if not, it is dropped immediately before the register which corresponds to its `alloca` is zeroed. Otherwise the register is not zeroed and the `alloca` will be released automatically by DIVINE.

4.5.4 Terminating Loop Optimization

In DIVINE a loop will generate a state at least once an iteration. This is caused by the heuristic which makes sure that state generation terminates. However, if the loop performs no visible action and always terminates, it is possible to run it atomically. This way, the entire loop is merged into one action, which leads to further reduction of the state space size.

This reduction is not implemented yet; however, to implement it, it would be necessary to have the following components:

1. loop detection, this is possible using LLVM's `LoopAnalysis`;
2. termination analysis for LLVM loops, which requires recovery of loop condition from LLVM IR and should employ some existing termination detection heuristic;
3. pointer analysis to detect if the loop accesses any variable which is (or might be) accessible from other threads; it is also possible to use `__divine_is_private` to detect visibility dynamically, or combine these approaches.

4.5.5 Nondeterminism Tracking

DIVINE is an explicit state model checker and it does not handle data nondeterminism well. Nevertheless, data nondeterminism is often useful, for example to simulate input or random number generation by a variable which can have an arbitrary value from some range. The only way to simulate such nondeterminism in DIVINE is to enumerate all the possibilities explicitly, using `__divine_choice`. This, of course, can lead to a large state space, as it causes branching of the size given by the argument of `__divine_choice`.

Nevertheless, for small domains, this handling of nondeterminism is quite efficient, as it does not require any symbolic data representation. This way, `__divine_choice` is used for example to simulate the failure of `malloc`: `malloc` can return a null pointer if the allocation is not possible and DIVINE simulates this in such a way that any call to `malloc` nondeterministically branches into two possibilities; either the `malloc` succeeds and returns memory, or it fails. Similarly, weak memory model simulation (Section 4.4) uses nondeterministic choice to determine which entry of the store buffer should be flushed.

For the verification of real-world programs, it is useful to be able to constrain nondeterminism which can occur in them, for example as a result of a call of the `rand` function, which returns a random number from some interval, usually from 0 to $2^{31} - 1$. Such a nondeterminism is too large to be handled explicitly. Nevertheless, it often occurs in patterns like `rand() % N` for some fixed and usually small number `N`. In these cases, it is sufficient to replace `rand() % N` with `__divine_choice( N )` which might be tractable for sufficiently small values of `N`.

To automate this replacement, at least in some cases an LLVM pass which tracks nondeterministic values and constraints the nondeterministic choice to the smallest possible interval can be created. A very simple implementation of such a pass which tracks nondeterminism only inside one function and recognizes two patterns, a cast to `bool` and modulo constant number, can be found in `lart/svcomp/svcomp.cpp`, class `NondetTracking`. For a more

complete implementation, limited symbolic execution of the part of the program which uses the nondeterministic value could be used. This version is not implemented yet.

4.6 Transformations for SV-COMP 2016

SV-COMP is a competition of software verifiers associated with the TACAS conference [12]. It provides a set of benchmarks in several categories, the benchmarks being written in C. DIVINE is participating in SV-COMP 2016 in the concurrency category which contains several hundred of short parallel C programs. Some of these programs have an infinite state space (usually infinite number of threads), or use nondeterministic data heavily and therefore are not tractable by DIVINE. There are, however, many programs which can be verified by DIVINE, with some minor tweaks.

In order to make it possible to verify the SV-COMP programs with DIVINE, they have to be pre-processed, as they use some SV-COMP-specific functions and rely on certain assumptions about the semantics of C which are not always met when C is compiled into LLVM.

1. The benchmark is compiled using `divine compile`.

2. Using LART, atomic sections used in SV-COMP are replaced with DIVINE's atomic sections which use `__divine_interrupt_mask`.

3. Using LART, all reads and writes to global variables defined in the benchmark are set to be volatile. This is done because SV-COMP models often contain undefined behaviour, such as concurrent access to non-volatile, non-atomic variables, which could be optimized improperly for SV-COMP. This pass actually hides errors in SV-COMP benchmarks; nevertheless, it is necessary, since SV-COMP benchmarks assume any use of shared variable will cause a load from it, which is not required by the C standard.

4. LLVM optimizations are run, using LLVM `opt` with `-Oz` (optimizations for binary size).

5. Nondeterminism tracking (Section 4.5.5) is used.

6. LART is used to disable nondeterministic failures in `malloc` since SV-COMP assumes that `malloc` never fails.

7. Finally, DIVINE is run on the program with Context-Switch-Directed Reachability [40] and assertion violations are reported if there are any. Other errors are not reported.

With these transformations, DIVINE is expected to score more than 900 points out of 1222 total in the concurrency category. A report which describes our approach is to appear in TACAS proceedings [41]. The implementation of these transformations can be found in the `lart/svmcomp/` directory.

Chapter 5

Results

In this chapter we will evaluate the transformations proposed in Chapter 4. All measurements were done on Linux on `x86_64` machines with enough memory to run verification of the program in question. All numbers are taken from DIVINE's report (`--report` was passed to the `verify` command). Number of states is `States-Visited` from the report, which is the number of unique states in the state space of the program; memory usage is `Memory-Used` from the report, which is the peak of the total memory used during the verification. All measurements were performed with the lossless tree compression enabled (`--compression`) [36] and, unless explicitly stated otherwise, the default setting of $\tau+$ reduction (which includes the changes described in Section 4.1.3).

Please note that the results in cases when the property does not hold depend on the timing and number of processors used for the evaluation. To make these results distinguishable, they are marked with dagger (†) superscript. Programs used in the evaluation are described in Table 5.1.

5.1 Extensions of $\tau+$ Reduction

Table 5.2 shows state space sizes of several models with the original $\tau+$ reduction and with the extensions described in Section 4.1.3. It also includes state space sizes in DIVINE 3.3, which is the version before any modification described in this thesis. While both DIVINE 3.3 and the new version with the original reduction implement the same reduction strategy, the numbers can differ because of bugs which were fixed since DIVINE 3.3. The first bug is that DIVINE 3.3 never considered `memcpy` to be a visible operation, which could cause some runs to be missed; with this bug fixed, the state space size can grow. The second bug is that if a visible instruction is at the beginning of a basic block, DIVINE 3.3 emitted a state immediately after this instruction; fixing this bug could cause the state space size to decrease. Finally, there was a bug in the calculation of visibility of a memory location; this information was improperly cached even over operations which could change the value.

`simple`	A program similar to the one in Figure 4.7; two threads, each of them reads a value written by the other one. An assertion violation can be detected with total store order. Written in C++; does not use C++11 atomics.
`peterson`	A version of the well-known Peterson's mutual exclusion algorithm; valid under sequential consistency, not valid under total store order or any more relaxed model. Written in C++, no C++11 atomics.
`fifo`	A fast communication queue for producer-consumer use with one producer and one consumer. This queue is used in DIVINE when running in a distributed environment. The queue is designed for `x86`; it is correct unless stores can be reordered. Written in C++, the queue itself does not use C++11 atomics, the unit test does use one relaxed (*monotonic*) atomic variable.
`fifo-at`	A modification of `fifo` which uses C++11 atomics to ensure it works with memory models more relaxed than TSO.
`fifo-bug`	An older version of `fifo` which contains a data race.
`hs-`T`-`N`-`E	A hight-performance, lock-free shared memory memory hash table used in DIVINE in shared memory setup [9]. Written in C++, uses C++11 atomics heavily, mostly with the *sequentially consistent* ordering. This model is parametrized; T is the number of threads, N is the number of elements inserted by each thread (elements inserted by each thread are distinct), and E is the number of extra elements which are inserted by two threads.
`pt-rwlock`	A test for a reader-writer lock in C.
`collision`	A collision avoidance protocol written in C++, described in [20].
`lead-dkr`	A leader election algorithm written in C++, described in [16].
`elevator2`	This model is a C++ version of the elevator model from the BEEM database [28]. It is a simulation of elevator planning.

Table 5.1: Description of programs used in the evaluation.

Name	DIVINE 3.3	Old	+ Control Flow	+ Indep. Load	New	Reduction
`fifo`	1.84 k	1.79 k	1.77 k	793	**791**	2.32×
`fifo-bug`	6.64 k†	6.61 k†	6.45 k†	2.88 k†	**2.88 k**†	2.31×
`lead-dkr`	29.9 k	132 k	132 k	58.2 k	**58.1 k**	0.51×
`collision`	3.06 M	3.28 M	3.28 M	1.96 M	**1.96 M**	1.56×
`pt-rwlock`	14.2 M	14.2 M	8.66 M	6.54 M	**4.48 M**	3.18×
`elevator2`	18.6 M	18.2 M	18.2 M	17.7 M	**17.7 M**	1.05×
`hs-2-1-0`	*error*	1.88 M	1.87 M	1.01 M	**891 k**	2.1×
`hs-2-1-1`	*error*	2.87 M	2.87 M	1.51 M	**1.34 M**	2.14×
`hs-2-2-2`	*error*	4.99 M	4.99 M	2.62 M	**2.33 M**	2.14×

Table 5.2: Evaluation of the improved $\tau+$ reduction. DIVINE 3.3 is used as a reference, as it does not include any changes described in this thesis. *Old* corresponds to the original $\tau+$ reduction with several bugfixes, *+ Control Flow* includes control flow loop detection optimization, *+ Indep. Load* includes independent loads optimization, *New* includes both optimizations. DIVINE 3.3 was not able to verify the hash set benchmarks.

We can see in the table that in all but one case, the extended $\tau+$ reduction performs better than DIVINE 3.3 and in all cases it performs better than the implementation of the original reduction in the new version of DIVINE. The one exception is `lead-dkr`; the reason is that this program uses `memcpy` heavily and therefore is affected by the bug fix. If we consider the fixed implementation as the baseline for `lead-dkr`, the new reduction represents a 2.27× improvement. Overall, the improvement was 1.05× to 3.18× for benchmarked models, which is a good improvement on the already heavy reduction of the original $\tau+$. We can also see that the independent loads optimization has a higher impact than the control flow loop detection optimization, but the latter still provides a measurable improvement (up to 1.5× reduction).

5.2 Weak Memory Models

We evaluated relaxed memory models on the same benchmarks as in [42] and additionally on a unit test for a concurrent hash table (`hs-2-1-0`). We used Context-Switch-Directed-Reachability algorithm [40] in all weak memory model evaluations, as it tends to find bugs in programs with weak memory models faster (it explores runs with fewer context switches and therefore fewer store buffer flushes earlier).

Table 5.3 shows state space sizes for programs with weak memory model simulation and compares them to the state space size of the original program. We can see that the size increase varies widely, but the increase is quite large,

Name	SC	TSO			TSO: Size Increase		
	-	1	2	3	1	2	3
`simple`	127	3.45 k†	5.97 k†	15.7 k†	27.1×	47×	123×
`peterson`	703	21.8 k†	53.4 k†	55.7 k†	31.1×	76×	79.2×
`fifo`	791	14.9 k	35.9 k	48.8 k	18.8×	45.3×	61.7×
`fifo-at`	717	39.5 k	167 k	497 k	55.1×	232×	693×
`fifo-bug`	1.61 k†	11.3 k†	44.2 k†	68.7 k†	7.01×	27.4×	42.6×
`hs-2-1-0`	891 k	250 M	–	–	281×	–	–

Name	SC	STD			STD: Size Increase		
	-	1	2	3	1	2	3
`simple`	127	3.52 k†	8.07 k†	23.6 k†	27.7×	63.6×	186×
`peterson`	703	22 k†	56.3 k†	69.8 k†	31.3×	80.1×	99.3×
`fifo`	791	18.3 k	15.6 k†	23 k†	23.1×	19.8×	29.1×
`fifo-at`	717	53.5 k	256 k	1.07 M	74.6×	357×	1489×
`fifo-bug`	1.61 k†	12.1 k†	14.1 k†	21.1 k†	7.53×	8.78×	13.1×
`hs-2-1-0`	891 k	251 M	–	–	282×	–	–

Table 5.3: A summary of the number of states in the state space for different weak memory simulation settings. The first line specifies the memory model (*SC* = Sequential Consistency, that is no transformation, *TSO* = Total Store Order, *STD* = the LLVM memory model). The second line gives store buffer size.

anywhere from 7× to 282× for a store buffer with only one slot. We can also see that the difference between total store order and more relaxed memory models is not as significant as the store buffer size increase, suggesting that there is still room for optimization of the TSO simulation. Benchmark `hs-2-1-0` shows that the weak memory model simulation is not yet easily applicable to more complex real-world code; in this case, the verification required 31.1 GB of memory and almost half a day of runtime on 48 cores, while larger versions of this model did not fit into a 100 GB memory limit. Nevertheless, for smaller real-world tests, such as `fifo`, the weak memory model simulation can be used even on a common laptop.

5.2.1 Effects of Optimizations

The optimizations described in Section 4.4.6 were first evaluated in the context of the TSO memory model simulation presented in [42]. The results of this initial evaluation can be seen in Table 5.4. This evaluation does not include any changes in $\tau+$ reduction. We can see that the effects of the optimizations are significant, especially for private loads optimization.

The same optimizations were evaluated in the final version of the transformation, these results can be seen in Table 5.5. Please note that while the

Name	MEMICS	+ Load Private		+ Locals	
`fifo-1`	44 M	5.6 M	7.9×	1.2 M	36×
`fifo-2`	338 M	51 M	6.6×	11 M	30×
`fifo-3`	672 M	51 M	13×	11 M	60×
`simple-1`†	538 k	19 k	28×	11 k	48×
`peterson-2`†	103 k	40 k	2.6×	24 k	4.1×
`pt_mutex-2`	1.6 M	12 k	135×	7.5 k	216×

Table 5.4: Effects of private load optimization and local private variable optimization on the implementation from [42]. The number after model name is the store buffer size. The *MEMICS* column shows the state space size for the original transformation, *+ Load Private* shows the state space size and reduction over *MEMICS* for the optimization which bypasses store buffers for thread-private memory locations; finally, *+ Locals* also includes the optimization which does not transform manipulations with local variables which do not escape the scope of the function.

original transformation bypassed store buffers for thread-private stores, the version proposed in this work does not, as this optimization is not correct for the LLVM memory model. Nevertheless, the new version performs an order of magnitude better in all cases, both thanks to enhanced state space reductions and a more efficient implementation.[1]

We can see that the effect of the optimizations of the store buffer implementation varies significantly, but overall the improvement is an approximately three-fold reduction, with a peak at `fifo-at` which is reduced up to several hundred times. This suggests that these reductions can have stronger effects on bigger programs. However, due to time and resource constraints, we were not able to verify this hypothesis with other large programs.

Table 5.6 shows the effects of extended $\tau+$ reduction on the LLVM memory model simulation. We can see that while the overall reduction is similar to the effects of the improved $\tau+$ reduction on programs without the memory model simulation, the reason is different. In this case, the improvement is thanks to the improved control flow cycle detection mechanism and the independent loads optimization has no effect. The cause is that `load`, `store`, and `fence` instructions are replaced with calls, and without the control flow cycle detection improvement it was not possible to perform two calls to the same function on one edge in the state space.

[1] Namely, the flusher thread is now implemented in such a way that is is guaranteed that if the store buffer associated with it contains a single entry and this entry is flushed, the resulting state of the flusher thread will be the same as the state of the flusher thread before the first entry is inserted into the store buffer.

Name	No opt.	+ Locals		+ Load Private	
simple-tso-1†	16.5 k	16.5 k	1×	**3.45 k**	4.79×
simple-tso-2†	29.3 k	29.3 k	1×	**5.97 k**	4.9×
simple-tso-3†	42 k	42 k	1×	**15.7 k**	2.68×
simple-std-1†	16.6 k	16.6 k	1×	**3.52 k**	4.7×
simple-std-2†	29.8 k	29.8 k	1×	**8.07 k**	3.69×
simple-std-3†	51.6 k	51.6 k	1×	**23.6 k**	2.18×
peterson-tso-1†	60 k	59.9 k	1×	**21.8 k**	2.75×
peterson-tso-2†	149 k	149 k	1×	**53.4 k**	2.79×
peterson-tso-3†	144 k	144 k	1×	**55.7 k**	2.59×
peterson-std-1†	60.2 k	60.2 k	1×	**22 k**	2.74×
peterson-std-2†	154 k	154 k	1×	**56.3 k**	2.72×
peterson-std-3†	168 k	168 k	1×	**69.8 k**	2.41×
fifo-tso-1	43.5 k	43.5 k	1×	**14.9 k**	2.92×
fifo-tso-2	110 k	110 k	1×	**35.9 k**	3.08×
fifo-tso-3	159 k	159 k	1×	**48.8 k**	3.26×
fifo-std-1	53.2 k	53.2 k	1×	**18.3 k**	2.91×
fifo-std-2†	45.5 k	46.4 k	0.98×	**15.6 k**	2.91×
fifo-std-3†	74.7 k	74.9 k	1×	**23 k**	3.25×
fifo-at-tso-1	314 k	121 k	2.58×	**39.5 k**	7.93×
fifo-at-tso-2	1.11 M	522 k	2.12×	**167 k**	6.64×
fifo-at-tso-3	5.46 M	1.46 M	3.75×	**497 k**	11×
fifo-at-std-1	433 k	166 k	2.6×	**53.5 k**	8.09×
fifo-at-std-2	32.2 M	739 k	43.6×	**256 k**	126×
fifo-at-std-3	–	2.8 M	–	1.07 M	–
fifo-bug-tso-1†	34.1 k	35.9 k	0.95×	**11.3 k**	3.02×
fifo-bug-tso-2†	141 k	141 k	1×	**44.2 k**	3.18×
fifo-bug-tso-3†	220 k	218 k	1.01×	**68.7 k**	3.21×
fifo-bug-std-1†	37.3 k	34.5 k	1.08×	**12.1 k**	3.07×
fifo-bug-std-2†	43.9 k	43.7 k	1×	**14.1 k**	3.1×
fifo-bug-std-3†	69.9 k	69.8 k	1×	**21.1 k**	3.31×

Table 5.5: Effects of private load optimization and local private variable optimization on the LLVM memory model simulation. The *No opt.* column includes none of the optimizations from Section 4.4.6, *+ Locals* does not instrument stores into local variables which do not escape the scope of the function, and *+ Load Private* also bypasses store buffers for loads from memory which is considered thread-private by DIVINE.

Name	Orig. $\tau+$	$\tau+$ Control Flow	$\tau+$ Indep. Loads	New	Reduction
simple-tso-1†	5.88 k	**3.45 k**	5.88 k	**3.45 k**	1.7×
simple-tso-2†	10.9 k	**5.97 k**	10.9 k	**5.97 k**	1.82×
simple-tso-3†	24.6 k	**15.7 k**	24.6 k	**15.7 k**	1.57×
simple-std-1†	5.9 k	**3.52 k**	5.9 k	**3.52 k**	1.68×
simple-std-2†	13.4 k	**8.07 k**	13.4 k	**8.07 k**	1.66×
simple-std-3†	32.4 k	**23.6 k**	32.4 k	**23.6 k**	1.37×
peterson-tso-1†	25.4 k	**21.9 k**	25.4 k	**21.9 k**	1.16×
peterson-tso-2†	63 k	**53.4 k**	63 k	**53.4 k**	1.18×
peterson-tso-3†	65.7 k	**55.7 k**	65.7 k	**55.7 k**	1.18×
peterson-std-1†	25.5 k	**22 k**	25.5 k	**22 k**	1.16×
peterson-std-2†	66.4 k	**56.4 k**	66.4 k	**56.4 k**	1.18×
peterson-std-3†	81.4 k	**69.8 k**	81.4 k	**69.8 k**	1.17×
fifo-tso-1	39.1 k	**14.9 k**	39.1 k	**14.9 k**	2.63×
fifo-tso-2	100 k	**35.9 k**	100 k	**35.9 k**	2.8×
fifo-tso-3	144 k	**48.8 k**	144 k	**48.8 k**	2.95×
fifo-std-1	48.3 k	**18.3 k**	48.3 k	**18.3 k**	2.64×
fifo-std-2†	41.6 k	**15.7 k**	41.6 k	**15.7 k**	2.65×
fifo-std-3†	66.1 k	**23.4 k**	66.1 k	**23.4 k**	2.83×
fifo-at-tso-1	114 k	**39.5 k**	114 k	**39.5 k**	2.89×
fifo-at-tso-2	490 k	**167 k**	490 k	**167 k**	2.94×
fifo-at-tso-3	1.39 M	**497 k**	1.39 M	**497 k**	2.79×
fifo-at-std-1	157 k	**53.5 k**	157 k	**53.5 k**	2.93×
fifo-at-std-2	704 k	**256 k**	704 k	**256 k**	2.75×
fifo-at-std-3	2.63 M	**1.07 M**	2.63 M	**1.07 M**	2.47×
fifo-bug-tso-1†	32.4 k	**11.9 k**	32.4 k	**11.9 k**	2.72×
fifo-bug-tso-2†	129 k	**45.5 k**	129 k	**45.5 k**	2.83×
fifo-bug-tso-3†	201 k	**66.4 k**	201 k	**66.4 k**	3.02×
fifo-bug-std-1†	33.7 k	**12.1 k**	33.7 k	**12.1 k**	2.79×
fifo-bug-std-2†	39.8 k	**14.2 k**	39.8 k	**14.2 k**	2.81×
fifo-bug-std-3†	63 k	**21.5 k**	63 k	**21.5 k**	2.93×

Table 5.6: Effects of the extended $\tau+$ reduction on the LLVM memory model simulation. *Reduction* shows the best achieved reduction.

5.3 LLVM IR Optimizations

We also evaluated transformations intended for state space reduction presented in Section 4.5. Namely, constant local variable elimination (Section 4.5.1, labelled *const* `alloca` in tables, the `paropt` pass in LART), constant global variable annotation (Section 4.5.2, labelled *const global* in tables, the `globals` pass in LART), register zeroing (Section 4.5.3, labelled *register zero* in tables, the `register` pass in LART) and an older version of register zeroing pass which zeroes only values of local variables (labelled `alloca` *zero* in tables, the `alloca` pass in LART). We also evaluated combinations of these optimizations. Please note that the order of the combination of these passes matters, constant local variable elimination must precede register (or local variable) zeroing. Constant global variable annotation does not interfere with any of the other reductions.

In Table 5.7 we can see the effect of these optimizations on the state space size. The only optimization with visible effect on state space size is constant local variable elimination. The effect of constant local variable elimination in not large and it is likely due to elimination of some registers which could have been used to distinguish otherwise equivalent states.

In Table 5.8 we can see the effect of the same optimizations on the memory required for verification with lossless compression of the state space. While these values are subject to some variation caused by the used compression technique,[2] we can see that memory-wise, the reductions have bigger effect. Namely, we can see that constant global variable annotation has a positive effect on memory requirements.

Finally, in Table 5.9, we can see the effect of the two most significant LART optimizations, constant local variable elimination and constant global variable annotation, on programs with the LLVM memory model simulation. We can see that these optimizations reduced state space size up to two times. We were not able to include more of hash set tests into this table due to their size.

[2] We use tree compression [36]. The efficiency of this reduction technique depends on the layout and the size of the state and on the patterns of changes in states. For this reason, even a slight variation in state layout can cause measurable difference in the compression ratio. Although these differences are much smaller than the overall effect of the compression, they are still visible in the table.

Name	`fifo`	`fifo-bug`	`collision`	`pt-rwlock`	`elevator2`
no LART	791	2876†	1.96 M	4.48 M	17.7 M
const `alloca`	791	2876†	1.96 M	**4.47 M**	**11.5 M**
const global	791	2876†	1.96 M	4.48 M	17.7 M
`alloca` zero	791	2876†	1.96 M	4.48 M	17.7 M
register zero	791	2876†	1.96 M	4.48 M	17.7 M
CA + CG	791	2876†	1.96 M	**4.47 M**	**11.5 M**
CA + CG + AZ	791	2876†	1.96 M	**4.47 M**	**11.5 M**
CA + CG + RZ	791	2876†	1.96 M	**4.47 M**	**11.5 M**
Reduction	1×	1×	1×	1×	1.54×

Name	`lead-dkr`	`hs-2-1-0`	`hs-2-1-1`	`hs-2-2-2`
no LART	58.1 k	891 k	1.34 M	2.33 M
const `alloca`	**43.4 k**	**875 k**	**1.32 M**	**2.29 M**
const global	58.1 k	891 k	1.34 M	2.33 M
`alloca` zero	58.1 k	904 k	1.36 M	2.35 M
register zero	58.1 k	891 k	1.34 M	2.33 M
CA + CG	**43.4 k**	**875 k**	**1.32 M**	**2.29 M**
CA + CG + AZ	**43.4 k**	888 k	1.33 M	2.31 M
CA + CG + RZ	**43.4 k**	**875 k**	**1.32 M**	**2.29 M**
Reduction	1.34×	1.02×	1.02×	1.02×

Table 5.7: Effects of LART optimizations on state space size. *Reduction* shows the best achieved reduction.

Name	fifo	fifo-bug	collision	pt-rwlock	elevator2
no LART	380 MB	380 MB†	**476 MB**	1.02 GB	1.24 GB
const `alloca`	335 MB	344 MB†	478 MB	1.01 GB	1.16 GB
const global	380 MB	382 MB†	501 MB	1.06 GB	**1.09 GB**
`alloca` zero	396 MB	394 MB†	513 MB	1.06 GB	1.39 GB
register zero	382 MB	379 MB†	501 MB	1.05 GB	1.37 GB
CA + CG	**331 MB**	**331 MB**†	476 MB	0.99 GB	1.27 GB
CA + CG + AZ	331 MB	331 MB†	476 MB	**0.99 GB**	1.27 GB
CA + CG + RZ	335 MB	335 MB†	477 MB	0.99 GB	1.26 GB
Reduction	1.15×	1.15×	1×	1.03×	1.14×

Name	lead-dkr	hs-2-1-0	hs-2-1-1	hs-2-2-2
no LART	644 MB	1.14 GB	1.15 GB	1.21 GB
const `alloca`	346 MB	559 MB	**823 MB**	883 MB
const global	385 MB	916 MB	923 MB	986 MB
`alloca` zero	398 MB	948 MB	969 MB	1.02 GB
register zero	414 MB	914 MB	925 MB	983 MB
CA + CG	337 MB	**547 MB**	826 MB	**853 MB**
CA + CG + AZ	**334 MB**	548 MB	831 MB	853 MB
CA + CG + RZ	355 MB	547 MB	827 MB	853 MB
Reduction	1.93×	2.14×	1.42×	1.45×

Table 5.8: Effects of LART optimizations on memory required for verification. *Reduction* shows the best achieved reduction.

Name	No LART	Const `alloca`		+ Const Global	
`simple-tso-1`†	3.45 k	**3.36 k**	1.03×	3.36 k	1.02×
`simple-tso-2`†	5.97 k	5.86 k	1.02×	**5.79 k**	1.03×
`simple-tso-3`†	15.7 k	15.4 k	1.02×	**15.4 k**	1.02×
`simple-std-1`†	3.52 k	**3.38 k**	1.04×	3.38 k	1.04×
`simple-std-2`†	8.07 k	7.51 k	1.07×	**7.45 k**	1.08×
`simple-std-3`†	23.6 k	19.7 k	1.2×	**19.4 k**	1.21×
`peterson-tso-1`†	21.8 k	**10.8 k**	2.02×	11 k	1.99×
`peterson-tso-2`†	53.4 k	33 k	1.62×	**32.6 k**	1.64×
`peterson-tso-3`†	55.7 k	**36 k**	1.55×	36.4 k	1.53×
`peterson-std-1`†	22 k	11 k	2×	**10.8 k**	2.04×
`peterson-std-2`†	56.3 k	35.6 k	1.58×	**35.5 k**	1.59×
`peterson-std-3`†	69.8 k	46.2 k	1.51×	**46.1 k**	1.51×
`fifo-tso-1`	14.9 k	12.7 k	1.17×	**12.7 k**	1.17×
`fifo-tso-2`	35.9 k	30.9 k	1.16×	**30.9 k**	1.16×
`fifo-tso-3`	48.8 k	42.1 k	1.16×	**42.1 k**	1.16×
`fifo-std-1`	18.3 k	15.7 k	1.17×	**15.7 k**	1.17×
`fifo-std-2`†	15.6 k	**13.2 k**	1.18×	13.6 k	1.15×
`fifo-std-3`†	23 k	**19.6 k**	1.17×	19.9 k	1.16×
`fifo-at-tso-1`	39.5 k	39.3 k	1.01×	**39.3 k**	1.01×
`fifo-at-tso-2`	167 k	166 k	1×	**166 k**	1×
`fifo-at-tso-3`	497 k	496 k	1×	**496 k**	1×
`fifo-at-std-1`	53.5 k	53.1 k	1.01×	**53.1 k**	1.01×
`fifo-at-std-2`	256 k	254 k	1.01×	**253 k**	1.01×
`fifo-at-std-3`	1.07 M	1.05 M	1.01×	**1.05 M**	1.01×
`fifo-bug-tso-1`†	11.3 k	10.2 k	1.11×	**10.1 k**	1.12×
`fifo-bug-tso-2`†	44.2 k	**38.6 k**	1.15×	38.8 k	1.14×
`fifo-bug-tso-3`†	68.7 k	**57.2 k**	1.2×	57.7 k	1.19×
`fifo-bug-std-1`†	12.1 k	10.6 k	1.15×	**10.4 k**	1.17×
`fifo-bug-std-2`†	14.1 k	12.4 k	1.14×	**11.9 k**	1.19×
`fifo-bug-std-3`†	21.1 k	**17.8 k**	1.18×	17.9 k	1.18×
`hs-2-1-0-tso-1`	250 M	184 M	1.36×	**184 M**	1.36×

Table 5.9: Results of weak memory model examples with LART optimizations.

Chapter 6

Conclusion

We showed that LLVM transformations are a useful preprocessing step for model checking of real-world programs, namely in verification of C and C++ using the DIVINE explicit-state model checker. We showed that LLVM transformations are usable in many ways, both to extend verifier's abilities and to decrease the size of the verification problem.

We proposed and implemented an instrumentation which adds under-approximation of LLVM memory models into a program in such a way that the result can be verified with DIVINE. This transformation allows verification of programs under the LLVM memory model on an unmodified verifier which assumes sequential consistency. This transformation extends the one presented in [42] in a number of ways, namely it supports parametrized memory models, including memory models weaker than total store order, it fully supports LLVM atomic instructions (and therefore C++11 atomic library), and it is up to several orders of magnitude more efficient than the version in [42]. Despite the state space explosion, which is even more pronounced for programs with relaxed memory models, we were able to verify several benchmarks, including unit tests of real-world data structures.

We also showed the usefulness of LLVM transformations on the case of annotated atomic functions, and on adaptation of SV-COMP benchmarks to DIVINE.

In the case of state space reductions, we proposed the use of LLVM optimizations which do not change the behaviour of parallel programs and do not increase state space size. We proposed and implemented a few such optimizations and evaluated them. We showed that some of these transformations, namely lifting of local variables into registers, can reduce state space size and memory requirements of DIVINE.

While working on this thesis, we also uncovered a bug in the implementation of DIVINE's $\tau+$ state space reductions and found cases in which these reductions could be improved. These improvements were implemented and evaluated and turned out to have a significant impact.

The proposed extensions, namely instrumentation for weak memory models, as well as some of the reduction techniques, will be included in the next version of DIVINE.

6.1 Future Work

There is a lot of future work in the field of LLVM transformations as a preprocessing step for verification of real-world code and we would like to continue to work in this field.

Efficient verification with weak memory models is still somewhat problematic and we believe that there is still room for improvement of the transformation technique. Namely, static detection of thread-local memory locations, for example using pointer analysis, could prove to be useful in reducing the state space size of programs with weak memory model simulation, since accesses to such memory locations need not be instrumented with memory model simulation. The results also show that the difference between total store order and partial store order simulation is not as big as expected, which suggests that the total store order simulation could be improved.

Further extensions of DIVINE's abilities by using LLVM transformations are also a topic for future work. One such possibility is the use of abstractions which was proposed in [33]; another is integration of the control-explicit-data-symbolic approach to model checking [3] into DIVINE with the help of LLVM transformations.

Finally, more advanced optimization techniques should be evaluated in the field of state space reductions. One example of such a technique is dealing with control flow loops which do not cause infinite loops in the program. Also, slicing, static partial order reduction, and symmetry reduction could be useful for state space reduction.

Appendix A

Archive Structure and Compilation

A.1 Archive Structure

The archive submitted with this thesis contains the sources of the thesis itself and a Darcs repository with DIVINE. The DIVINE repository is a snapshot of `http://paradise.fi.muni.cz/~xstill/dev/divine/next-xstill`, taken at the time of submission of this thesis. You can use the command `darcs log -is` in the `divine` subdirectory to browse the changes to DIVINE. Since the primary aim of this thesis are LLVM transformations, most of the implementation was done in LART; patches concerning LART are prefixed with `LART`, they can be listed by `darcs log -is --match 'name LART:'`.

A.2 Compilation and Running of DIVINE and LART

A.2.1 Prerequisites

In order to compile DIVINE and LART, it is necessary to have an up-to-date Linux distribution with a C++14 capable compiler. The compilation was tested with GCC 5.3.0 and Clang 3.7.0, both using libstdc++ 5.3.0 as the C++ standard library. Furthermore, it is necessary to have LLVM 3.7.0, including development libraries, Clang 3.7.0, and CMake.

A.2.2 Compilation

It should be sufficient to run the following commands in the root directory of the archive; please pay attention to the output of `configure`, to see if it was able to find LLVM and Clang:

```
cd divine
./configure
```

```
cd _build
make lart divine
ls tools
```

The last command should show that there are binaries `divine` and `lart` in `tools` subdirectory.

A.2.3 Compilation of Program for DIVINE

An input for DIVINE is an LLVM bitcode file which can be obtained from the source using the `divine compile` command, for example:

```
./tools/divine compile model.cpp --cflags=-std=c++11
```

Please refer to `divine compile --help` for more details.

A.2.4 Running LART

The basic usage of LART is: `lart <input> <output> [<pass> [...]]`, the `input` and `output` are LLVM bitcode files, each `pass` is a LART pass; a list of available passes can be seen by running `lart` without any parameters.

For example bitcode in `model.bc` can be instrumented with weak memory models (Section 4.4) with store buffer limited to 2 entries with the following command:

```
# this is unrestricted LLVM memory model
./tools/lart model.bc model-wm.bc weakmem::2
# Total Store Order:
./tools/lart model.bc model-wm.bc weakmem:tso:2
```

A.2.5 Running DIVINE

The model can be verified by DIVINE using the `divine verify` command; this command expects a model name and optionally several parameters such as the algorithm, the reductions to be used, and the property to be verified. Among the most important options is `--compression`, which enables lossless tree compression, which vastly improves memory efficiency of DIVINE.

```
# run DIVINE with default property (safety) on model-wm.bc
./tools/divine verify --compression model-wm.bc
# verify only assertion safety
./tools/divine verify --compression model-wm.bc -p assert
# use Context Switch Directed Reachability algorithm
./tools/divine verify --csdr --compression model-wm.bc
# verify exclusion LTL property (specified in the model)
./tools/divine verify --compression model-wm.bc -p exclusion
```

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Pearson Education, Inc, 2006. ISBN: 0321486811.

[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. Madrid, Spain: ACM, 2010, pp. 7–18. ISBN: 9781605584799. DOI: 10.1145/1706299.1706303.

[3] Jiří Barnat, Petr Bauch, and Vojtěch Havel. "Model Checking Parallel Programs with Inputs". In: *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*. Feb. 2014, pp. 756–759. DOI: 10.1109/PDP.2014.44.

[4] Jiří Barnat, Luboš Brim, Milan Češka, and Petr Ročkai. "DiVinE: Parallel Distributed Model Checker". In: *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*. IEEE. Sept. 2010, pp. 4–7. DOI: 10.1109/PDMC-HiBi.2010.9.

[5] Jiří Barnat, Luboš Brim, and Vojtěch Havel. "LTL Model Checking of Parallel Programs with Under-Approximated TSO Memory Model". In: *Application of Concurrency to System Design (ACSD)*. IEEE, 2013, pp. 51–59. DOI: 10.1109/ACSD.2013.8.

[6] 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". English. In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 863–868. ISBN: 9783642397981. DOI: 10.1007/978-3-642-39799-8_60.

[7] Jiří Barnat, Luboš Brim, and Petr Ročkai. "On-the-fly Parallel Model Checking Algorithm that is Optimal for Verification of Weak LTL Properties". In: *Science of Computer Programming* 77.12 (Oct. 2012), pp. 1272–1288. ISSN: 0167-6423. DOI: 10.1016/j.scico.2011.03.001.

[8] Jiří Barnat, Ivana Černá, Petr Ročkai, Vladimír Štill, and Kristína Zákopčanová. "On Verifying C++ Programs with Probabilities". In: *to appear in ACM Symposium on Applied Computing*. 2016. ISBN: 9781450337397. DOI: 10.1145/2851613.2851721.

[9] Jiří Barnat, Petr Ročkai, Vladimír Štill, and Jiří Weiser. "Fast, Dynamically-Sized Concurrent Hash Table". English. In: *Model Checking Software (SPIN 2015)*. Ed. by Bernd Fischer and Jaco Geldenhuys. Vol. 9232. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 49–65. ISBN: 9783319234038. DOI: 10.1007/978-3-319-23404-5_5.

[10] Jiří Barnat, Petr Ročkai, Vladimír Štill, and Jiří Weiser. *DIVINE: Model Checking for Everyone*. 2016. URL: http://divine.fi.muni.cz/ (visited on 01/09/2016).

[11] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. "UPPAAL — a Tool Suite for Automatic Verification of Real-time Systems". English. In: *Hybrid Systems III*. Ed. by Rajeev Alur, ThomasA. Henzinger, and EduardoD. Sontag. Vol. 1066. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 232–243. ISBN: 9783540611554. DOI: 10.1007/BFb0020949.

[12] Dirk Beyer. *Competition on Software Verification (SV-COMP), TACAS 2016*. 2016. URL: http://sv-comp.sosy-lab.org/2016/ (visited on 01/09/2016).

[13] ISO C Standards Committee. *Programming languages — C, Committee Draft N1570*. Tech. rep. ISO IEC JTC1/SC22/WG14, 2011.

[14] ISO C++ Standards Committee. *Standard for Programming Language C++. Working Draft N3337*. Tech. rep. ISO IEC JTC1/SC22/WG21, 2012.

[15] David L. Dill. "The Murphi Verification System". In: *Proceedings of the 8th International Conference on Computer Aided Verification*. CAV '96. London, UK, UK: Springer-Verlag, 1996, pp. 390–393. ISBN: 3540614745. URL: http://dl.acm.org/citation.cfm?id=647765.735832.

[16] Danny Dolev, Maria M. Klawe, and Michael Rodeh. "An O(n log n) Unidirectional Distributed Algorithm for Extrema Finding in a Circle". In: *J. Algorithms* (1982), pp. 245–260.

[17] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0262032708.

[18] 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.

[19] IEEE Portable Applications Standards Committee. *Std 1003.1c-1995 Standard for Information Technology — Portable Operating System Interface (POSIX) — System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE Computer Society Press, 1995.

[20] Henrik Ejersbo Jensen, Jensen Kim, Kim Guldstrand Larsen, and Arne Skou. "Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL". In: *Proceedings of the 2nd International Workshop on the SPIN Verification System*. 1996.

[21] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. "LTSmin: High-Performance Language-Independent Model Checking". English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015, pp. 692–707. ISBN: 9783662466803. DOI: 10.1007/978-3-662-46681-0_61.

[22] Daniel Kroening and Michael Tautschnig. "CBMC – C Bounded Model Checker". English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 389–391. ISBN: 9783642548611. DOI: 10.1007/978-3-642-54862-8_26.

[23] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization". MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002. URL: http://llvm.org/pubs/2002-12-LattnerMSThesis.html.

[24] Chris Lattner. *The LLVM Compiler Infrastructure Project*. 2016. URL: http://llvm.org/ (visited on 01/09/2016).

[25] Alexander Linden and Pierre Wolper. "A Verification-Based Approach to Memory Fence Insertion in PSO Memory Systems". English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Nir Piterman and Scott A. Smolka. Vol. 7795. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 339–353. ISBN: 9783642367410. DOI: 10.1007/978-3-642-36742-7_24.

[26] Alexander Linden and Pierre Wolper. "An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models". English. In: *Model Checking Software*. Ed. by Jaco van de Pol and Michael Weber. Vol. 6349. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 212–226. ISBN: 9783642161636. DOI: 10.1007/978-3-642-16164-3_16.

[27] Seungjoon Park and David L. Dill. "An Executable Specification and Verifier for Relaxed Memory Order". In: *IEEE Trans. Comput.* 48.2 (Feb. 1999), pp. 227–235. ISSN: 0018-9340. DOI: 10.1109/12.752664.

[28] Radek Pelánek. "BEEM: Benchmarks for Explicit Model Checkers". English. In: *Model Checking Software*. Ed. by Dragan Bošnački and Stefan Edelkamp. Vol. 4595. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 263–267. ISBN: 9783540733690. DOI: 10.1007/978-3-540-73370-6_17.

[29] LLVM Project. *LLVM Atomic Instructions and Concurrency Guide*. 2016. URL: http://llvm.org/docs/Atomics.html (visited on 01/09/2016).

[30] LLVM Project. *Exception Handling in LLVM*. 2016. URL: http://llvm.org/docs/ExceptionHandling.html (visited on 01/09/2016).

[31] LLVM Project. *LLVM Language Reference Manual*. 2016. URL: http://llvm.org/docs/LangRef.html (visited on 01/09/2016).

[32] *RAII – cppreference.com*. 2016. URL: http://en.cppreference.com/w/cpp/language/raii (visited on 01/09/2016).

[33] Petr Ročkai. "Model Checking Software". Ph.D. Thesis. Masaryk University, Faculty of Informatics, Brno, 2015. URL: http://is.muni.cz/th/139761/fi_d/.

[34] 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)*. Vol. 7871. LNCS. Springer, 2013, pp. 1–15.

[35] Petr Ročkai, Jiří Barnat, and Luboš Brim. "Model Checking C++ with Exceptions". In: *Automated Verification of Critical Systems*. AVoCS 2014. 2014. DOI: 10.14279/tuj.eceasst.70.983.

[36] Petr Ročkai, Vladimír Štill, and Jiří Barnat. "Techniques for Memory-Efficient Model Checking of C and C++ Code". English. In: *Software Engineering and Formal Methods*. Ed. by Radu Calinescu and Bernhard Rumpe. Vol. 9276. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 268–282. ISBN: 9783319229683. DOI: 10.1007/978-3-319-22969-0_19.

[37] Pavel Šimeček. "DiVinE – Distributed Verification Environment". Master's Thesis. Masaryk University, Faculty of Informatics, Brno, 2006. URL: http://is.muni.cz/th/51636/fi_m/.

[38] Carsten Sinz, Florian Merz, and Stephan Falke. "LLBMC: A Bounded Model Checker for LLVM 's Intermediate Representation". English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Cormac Flanagan and Barbara König. Vol. 7214. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 542–544. ISBN: 9783642287558. DOI: 10.1007/978-3-642-28756-5_44.

[39] SPARC International, Inc., CORPORATE. *The SPARC architecture manual (version 9)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994. ISBN: 0130992275.

[40] Vladimír Štill, Petr Ročkai, and Jiří Barnat. "Context-Switch-Directed Verification in DIVINE ". English. In: *Mathematical and Engineering Methods in Computer Science*. Ed. by Petr Hliněný, Zdeněk Dvořák, Jiří Jaroš, Jan Kofroň, Jan Kořenek, Petr Matula, and Karel Pala. Vol. 8934. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 135–146. ISBN: 9783319148953. DOI: 10.1007/978-3-319-14896-0_12.

[41] Vladimír Štill, Petr Ročkai, and Jiří Barnat. "DIVINE: Explicit-State LTL Model Checker". In: *to appear in International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2016.

[42] Vladimír Štill, Petr Ročkai, and Jiří Barnat. "Weak Memory Models as LLVM-to-LLVM Transformations". In: *to appear in Mathematical and Engineering Methods in Computer Science*. 2015.