Faculty of Informatics, Masaryk University

Caching SMT Queries in SymDIVINE

Bachelor's thesis

Jan Mrázek

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

Scalability of automatic verification tools is a crucial factor for usability of such tools in practise. There are vast number of ways to improve it. Trading off space for time is one of a classic approaches to improving time efficiency of these tools, mainly used when constraints satisfiability checking plays a central role. Caching of quantifier-free Satisfiability Modulo Theories (SMT) queries is now widely used in the world of symbolic execution.

SymDIVINE is a tool for bit-precise control-explicit data-symbolic model checking of parallel C and C++ programs. Quantified SMT queries for multi-state equality decisions play a central role in SymDIVINE and take most of the verification time. Standard caching techniques do not work due to the quantification. In this thesis we propose *dependency-based caching* for quantified SMT queries, that are used in SymDIVINE. We also demonstrate integration of it in SymDIVINE and provide experimental evaluation on a diverse set of benchmarks.

Keywords

Formal Verification, C, C++, LLVM, Model Checking, SMT, Caching, Implementation

Acknowledgements

First, I would like to thank all the people in the ParaDiSe laboratory for providing such a nice half work-friendly and half procrastination-friendly environment. It was a pleasure to write this thesis with you! Namely, I would like to thank Jiří Barnat for providing me the opportunity to become a member of ParaDiSe laboratory and advising this thesis, and Vladimír Štill for the instant small consultations he provided me all the time.

I would like to thank Zuzana Baranová and Jana Mrázová for consulting the traps of the English language with me. Also, I would like to thank my family for supporting me and having enough patience with me.

Finally, I cannot forget to thank all the people who constantly reminded me to work. This thesis would not exist without you.

Contents

Chapter 1

Introduction

Validation and verification are one of the essential parts of the software development, as software bugs in a released product may become costly and degrade the overall rating of the vendor, or even cause harm to the users. Therefore, a lot of effort is usually put into this part of the development process even though it is one of the most time-consuming and most expensive part.

Testing is a widely adopted method in the industry, as it is quite simple and does not require any complex tools. Usually only a simple testing framework and an (automated) runtime environment is used. Even though this method is not sound (it cannot prove the absence of a bug), it performs quite well in practice during bug-finding in sequential code.

As multi-core CPUs are quite common today, multi-threaded software is required to fully utilise them. Even mobile devices, such as cellphones, feature multi-core CPUs and thus multi-threaded software is increasingly commonly produced today. This kind of software is hard to test due to the presence of non-determinism in thread scheduling. Two runs of the same program with the same input can lead to a different threads interleaving. This can cause distinct program behaviour for each run – so-called race conditions. Multi-threaded program tests are also affected by the scheduler's non-determinism, so it is possible to obtain two different test results for two test runs. These bugs are hard to find, as they can occur only in a single thread interleaving, which can be scheduled only in very specific circumstances. Observation of the program (e.g. run when under a debugger or run in a different environment) can also affect the scheduling and the bug might not occur.

A lot of effort has been put into the development of formal methods during the last few decades. These methods could replace testing and would help to find more bugs. There are methods like symbolic execution, bounded model checking and others, which are in general unsound; however, they can help with discovery of the hard-to-find bugs (e.g. integer overflow related bugs, memory safety etc.). On the other hand, there are methods like deductive verification, model-checking and others, which are sound, and, besides bug-finding, they can actually prove absence of a bug. Many of these methods also support multi-threaded programs, so it is possible to deterministically find bugs in multi-threaded software. This could be one of the motivations to replace testing by these methods in the industry.

Despite the promising features of the formal methods, they are not widespread in the industry and stay mainly within the academic interest. There are several reasons for that. First of all, techniques like deductive verification are not automatized and require qualified user interaction. On the other hand, the automatized techniques do not scale well to real world programs. Either they cannot take an unmodified code and process it (manual annotation is needed, all language features are not supported), or the verification needs an enormous amount of computing resources for real-world code. Many tools fail to verify real-world code due to substantial input data domain or complexity in control flow.

One of the tools that aim for verification of real-world parallel C/C++ code is SymDIVINE. This tool is a control-explicit data-symbolic model checker. Unlike explicit-state model checkers, it can handle non-deterministic input well. It allows user to take an unmodified C/C++ code with notation of input values and verify it for reachability or LTL properties. To alleviate state-space explosion caused by input values, a so-called set-based reduction is incorporated in this tool. The reduction is based on symbolic data representation, which heavily uses quantified bit-vector SMT queries to an SMT solver.

In this thesis we propose and implement new optimizations for SMT machinery in SymDIVINE in order to speed-up the verification task and thus help SymDIVINE to scale better. Our optimizations are based on caching of SMT queries.

The thesis is organised as follows: first, we a make an short overview of related topics in Chapter 2. Then we provide a detailed description of SymDIVINE's internals that are essential for our thesis in Chapter 3. In Chapter 4 we make an overview of the existing SMT caching solutions and propose a new one. The results of the experimental evaluation of our optimization are presented in Chapter 5. Chapter 6 summarizes our contribution and discusses future work.

Chapter 2

Preliminaries

2.1 Explicit-State Model Checking

Model checking is a formal method for verification of finite-space systems against given property [12]. This check can be performed fully automatically by a software tool. The core idea of this method is that the whole state-space of a system can be produced and explored. Usually the system is composed of multiple processes that can interact with each other. During the exploration of the system state-space, all interleavings are explored, so a detection of race conditions is possible. When a property is violated, model checker can produce a counter-example – a run that violates the property.

The specification can be expressed as a formula in a temporal logic, like LTL, CTL or CTL*. The system is traditionally specified in a special modelling language (e.g. ProMeLa in case of Spin [14], or DVE in case of DIVINE [3]). However, this method can be also be used for verification of computer programs, implying the existence of tools that take a program source code instead of a special modelling language. Examples of such tools are CBMC [17] or DIVINE, that can take a C or C++ code and verify it against the given property.

Explicit-state model checking considers all possible memory configurations of the system. This puts a restriction to verified programs – they can hardly read non-deterministic input values from the outside world (all possible valuations has to be enumerated). There are, however, methods like the control-explicit data-symbolic approach [2], on top of which is SymDIVINE build, that can help to solve this issue.

The limiting scalability factor of model checking is the so-called state-space explosion. With non-determinism in the system (the possibility of resource acquisition failure, scheduler in multi-threaded systems), the number of possible runs and consequent size of the state-space, grows exponentially. All modern model checkers involve techniques for state-space reduction – e.g. $\tau+$-reduction [23] in DIVINE.

2.2 Symbolic Execution

Symbolic execution [16] is another formal verification technique that, unlike model checking, primarily aims for verification of programs that can read non-deterministic input values. It does not usually handle parallel systems well. KLEE [10] is an example of a tool for symbolic execution of C and C++ programs.

Symbolic execution basically executes the program and, instead of obtaining concrete values of program variables, represents their values symbolically. When a branching on a non-deterministic value occurs, the computation is split into two paths – one where the branching condition was true, the other one where the branching condition was false. In each branch, a constraint for variable values is constructed. These constraints form a so-called path-condition. The exploration produces a symbolic execution tree.

Symbolic execution might not terminate, even on finite-state systems, if an infinite cycle in the system is present. Symbolic execution is quite wide-spread and there are many mutations of this technique, e.g. using concrete values to speed-up the process and using the symbolic part only for synthesis of new values during branching. Symbolic execution can also be used for automatic synthesis of tests for software [16].

2.3 LLVM

LLVM [18] is a compiler infrastructure. This infrastructure features tools for optimization and code generation that are independent of programming language and platform. This is achieved by definition of a custom intermediate representation – LLVM IR (also called LLVM bit-code). This representation is suitable for software verification tools as it can be easily interpreted and precisely reflects semantics of input value.

So-called compiler front-ends translate the input programming language (C/C++, Haskell, etc.) in the simplest possible way to LLVM IR. All further optimizations are performed on top of LLVM IR – each optimization is written as LLVM to LLVM transformation and thus optimization may be applied in arbitrary order, even multiple times. So-called back-ends performs the last step of compilation – the translation of LLVM IR to native code for a given platform.

LLVM IR is a static-single-assignment-based low level language similar to common assembly. It can be represented in three ways: a human-readable form (`.ll` file), a serialized form for fast machine handling (`.bc` file) or in the form of an in-memory C++ objects. There is a library that allows easy manipulation with all the forms of LLVM IR, which is a huge advantage for software verifiers, as it presents a way to easily take almost any input programming language relatively effortlessly. In the following text, we will shortly introduce the most important aspects of LLVM IR.

An LLVM program consists of modules [22]. A module contains functions, global variables and meta-data. Each module is a product of a single compilation unit or as a product of an LLVM linker. There are two basic identifiers in LLVM: global identifiers (variables, functions) denoted with `@` before the name and local identifiers (registers, labels) denoted with `%` before the name.

Each function consists of its header (name and parameters definitions) and a body. The body consists of basic blocks, with each basic block being a sequence of instructions. Basic blocks cannot contain any branching except the last instruction. A branching instruction can decide which basic block will be executed next. A function can use unlimited number of registers to store its data. These registers are in the SSA form. LLVM instructions operate strictly on registers, except `load`, `store`, `atomicrmw` and `cmpxchg` instructions, which can be used for memory manipulation. The address of a register cannot be taken. To obtain a variable, whose address can be taken, `alloca` instruction can be used. This instruction takes a size and a type and returns an address of a memory location. The memory is automatically freed when function returns (it is usually implemented as a stack allocation).

LLVM is a typed language. There are 4 main groups of types: integral types, floating point types, pointer types and composed types. Integral types can be any width and are denoted by `i` and its bit-width – e.g. `i32` for a typical integer or `i1` for boolean value. There are two floating point types – `float` and `double`. Pointer types are denoted in the same way as in C – e.g. `i32*` for a pointer to a 32-bit integer. Composed types can be arrays (`[16 x i32]`) or structures (`{i32, i8, double}`). Types in LLVM cannot be implicitly cast. To preform a cast, a special instruction is required.

2.4 Satisfiability Modulo Theories

There are many applications in computer science that can benefit from decision procedure for first-order logic formula satisfiability. Even though there are solvers for first-order logic [26], many applications do not require general first-order logic, but rather need satisfiability with respect to a given background-theory. This background theory usually fixes the domain and interpretation of predicate and function symbols. A background theory can usually yield a specialized, more effective decision procedure.

The research field concerned with satisfiability of formulae with respect to these theories is called *Satisfiability Modulo Theories* (SMT) [1]. There are many solvers for various theories: e.g. Z3 [19], CVC4 [4] or OpenSMT [8]. Much effort has been put into standardization of input to these solvers. This effort resulted in the SMTLib format [5] which specifies input language and theories. In our work, we are mainly interested in SMT for quantified and quantifier-free version of the FixedSizeBitVectors theory.

Chapter 3

SymDIVINE

In this chapter, we introduce SymDIVINE from the user point of view and then describe its internal architecture. For purposes of this thesis we focus mainly on selected parts of the LLVM interpreter in SymDIVINE, SMT data representation and the related machinery in SymDIVINE. We omit mainly technical details about LLVM interpretation and general optimizations. In many cases, we provide more in-depth description than [13], however, for others, we kindly refer to that thesis.

3.1 About the Tool

SymDIVINE is a tool for verification of real-world parallel C and C++ programs with non-deterministic inputs. It is being developed at ParaDiSe Laboratory, at Faculty of Informatics Masaryk University. It is distributed under MIT licence. Source code can be found at `https://github.com/yaqwsx/SymDIVINE`.

The tool is built on top of the LLVM framework in order to avoid the need of modelling and, at the same time, to achieve the precise semantics of C and C++ programming languages. SymDIVINE is motivated as an extension of purely explicit model checker DIVINE [3] that is capable of handling full parallel C/C++ programs without inputs. SymDIVINE shares the ideology of DIVINE – it aims for bitprecise[1] verification of parallel C and C++ programs without modification. To properly handle programs with inputs, SymDIVINE relies on control-explicit data-symbolic approach [2], which we detail in the next section.

The tool was originally presented in [13] as a generic platform for control-explicit data-symbolic state-space exploration. It provided a state generator from LLVM bitcode and allowed the user to specify a custom state format and an exploration algorithm. This set-up let the user to implement a wide

[1]All operations precisely keep the semantics of the original program. This is mainly of arithmetic concern – integers have limited bit-width and overflow on unsigned types is defined.

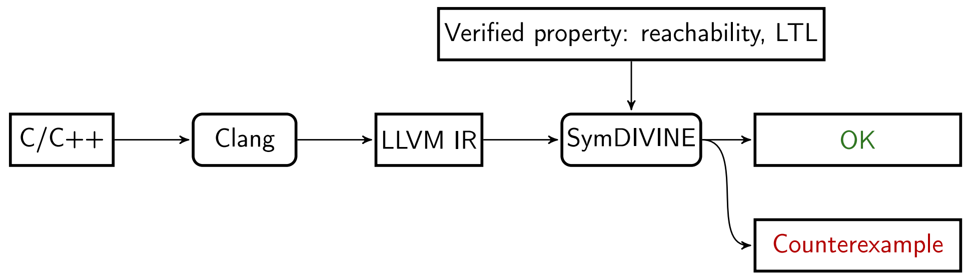

Figure 3.1: Typical verification workflow in the current version of SymDIVINE. It takes an LLVM bitcode and property specification on input and decides whether supplied property does hold or not. If not, a counterexample can be produced by the verification algorithm. Input bitcode is usually produced by the Clang compiler from C or C++ source code, however the user can obtain it differently.

variety of verification techniques – e.g. explicit-state model checking, symbolic execution or some kind of hybrid technique. Over the years, SymDIVINE transformed from a generic platform to an "out-of-box ready" verification tool [20] by providing predefined SMT-based state representation and implementation of algorithms for assertion safety and LTL properties checking. See Figure 3.1 for typical verification workflow in the current release [2]. Algorithms and store implementation provided in the current release were tested in practise and provide quite good performance. Nevertheless, the internal modular architecture was preserved, so SymDIVINE can still be used as a platform for user experiments.

3.1.1 Input Language Overview

SymDIVINE is designed to take an LLVM bitcode as an input language and thus support for C and C++ languages features is mainly reduced to support of the LLVM instruction set. In the current version SymDIVINE supports almost all LLVM instructions except of:

- instructions for symbolic pointer arithmetic,
- instructions for pointer casts[3],
- instructions for floating point arithmetic.

To verify a program using SymDIVINE, the input LLVM bitcode has to be self-contained – there must not be any call to functions that are not defined in

[2] Current version at the time of writing this thesis is v0.3. Release of this version is available at `https://github.com/yaqwsx/SymDIVINE/releases/tag/v0.3` and in the electronic archive submitted with this thesis.

[3] Simple integer conversions are supported, only the real 'bitcast' operation is not.

the bitcode. Behaviour of such functions is unknown to the tool and thus they cannot be verified. This also includes system calls to an underlying operating system. There are, however, a few exceptions, as SymDIVINE provides intrinsic implementation[4] of a subset of Pthread library [15] to support multi-threading and also a subset of functions defined in SV-COMP competition rules [7] to implement a notation for non-deterministic input.

The following functions from Pthread library are supported:

- `pthread_create`, `pthread_join` and `pthread_create` for thread manipulation,
- `pthread_mutex_lock` and `pthread_mutex_unlock` for mutex manipulation.

The following functions from SV-COMP notation are supported:

- `__VERIFIER_nondet_{type}` for modelling a non-deterministic input of a given type,
- `__VERIFIER_atomic_begin` and `__VERIFIER_atomic_end` for modelling atomic sections.

This means that the standard C and C++ library is supported if it is linked to the input program in the bitcode form and used functions do not call any system calls.

The current version of SymDIVINE does not support heap allocation, so the verified program is forbidden to call `malloc` or use the `new` operator. Likewise dynamic sized arrays from C99 cannot be handled in the current version of SymDIVINE.

3.2 Control-Explicit Data-Symbolic Approach

In the standard explicit-state model checking, the state-space graph of the verified programs is explored by an exhaustive enumeration of its states. SymDIVINE basically follows the same idea, but it employs the control-explicit data-symbolic approach to alleviate the state space explosion caused by the non-deterministic input values.

When an input read (`__VERIFIER_nondet_{type}` function is called) is interpreted by an explicit-state model checker, a new successor for every possible input value has to be produced. This causes a tremendous state-space explosion. It is worth noting that these states only differ in a single data field and thus the same instructions are further applied to all of them (only

[4]Behaviour of these functions is hard-coded in the interpreter and follows their specification.

branching or the `select` instruction can change the control flow). SymDIVINE can benefit from this fact. When SymDIVINE interprets non-deterministic read, only a single so-called *multi-state* is produced. The produced multi-state is composed of an explicit control flow location and a set of program's memory valuation.

A single multi-state can be viewed as a set of purely explicit states (so-called set-based reduction, we kindly refer to [13] for formal definition). As a result, multi-state space can bring up to exponential size (and memory) reduction compared to the explicit state-space of the same program with inputs. The model-checking algorithms present in SymDIVINE operate exclusively on multi-states. As the multi state-space is up to exponentially smaller and a single operation application to a multi-state corresponds to an application of the same operation to a set of explicit states, these algorithms can be much faster, even though handling multi-states is computationally more demanding. This is the key differentiation compared to the purely explicit approaches. To illustrate the effect of the set-based reduction, see an example of a bitcode and the corresponding explicit-state space and a multi-state space in Figure 3.2.

Moreover, if we provide a decision procedures for multi-state equality, it is possible to adopt existing explicit-state model checking algorithms. This allows easy implementation of standard automata-based LTL model-checking or perform safety analysis for non-terminating programs (provided that the multi-state space is finite).

To verify a real-world program, an efficient representation of the multi-states is needed. SymDIVINE is not linked to a given fixed format of multi-states and users can supply their own implementation (as described in detail in Section 3.3). During development of SymDIVINE several representations of multi-states were implemented and tested. This includes representation using binary decisions diagrams (BDD) or representation using SMT formulae. BDD representation performs quite well on artificial benchmarking programs containing no advanced arithmetic. SMT representation significantly outperformed the previous one on real-world programs with more arithmetic. Support for BDD representation was dropped in the current version and the SMT representation is the only one shipped. We describe this representation in detail in Section 3.4, as it an essential preliminary for our work presented in this thesis.

3.3 Internal Architecture

As we mentioned in the previous section, SymDIVINE was originally developed as a platform for creating custom tools for the control-explicit data-symbolic approach. Thus the internal structure is split into clearly-separated modules with fixed interface and each module can easily be replaced by another implementation. The whole tool is implemented in C++. Each module is

```
%a = call i32 @__VERIFIER_nondet_int()
%b = icmp sge i32 %a, 65535
br i1 %b, label %5, label %6
```

The code represents a simple LLVM program, where register `a` is initialized with a non-deterministic 32-bit integer, then is checked whether it is greater or equal to a given constant. The result of the check is stored to register `b` and used for branching.

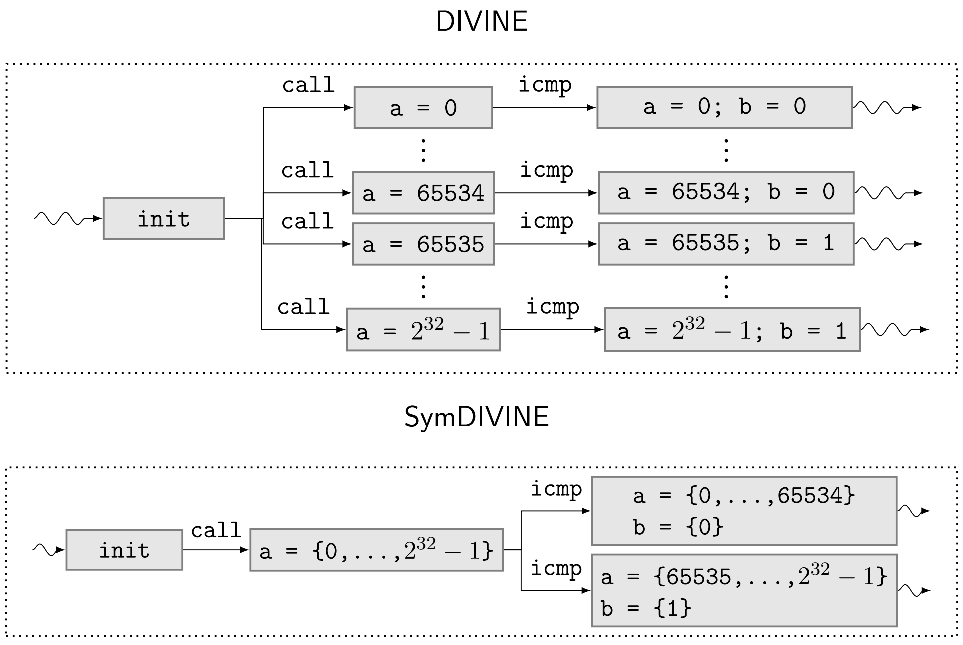

Figure 3.2: The figure compares state exploration in the explicit approach of DIVINE and in the control-explicit data-symbolic approach of SymDIVINE on an LLVM program example. From `init` state DIVINE explores states for every possible value of `a` (2^{32} values), hence exponentially expands the state space. In contrast, the SymDIVINE approach of symbolic representation generates only two different states. One where the condition on branching ($a \geq 65535$) is satisfied and the other one where the condition is violated.

represented by a class. Figure 3.3 illustrates components interaction. There are following main modules in SymDIVINE:

- the LLVM interpreter (responsible for multi-state generation from LLVM bitcode),
- data stores (implementation of multi-state representation),

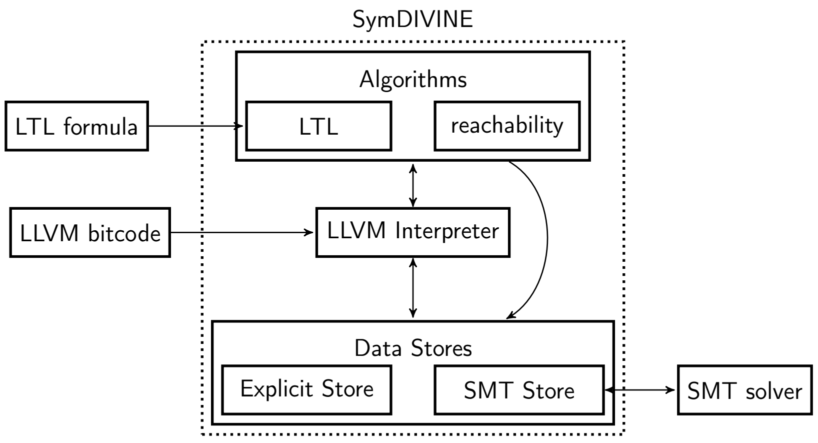

Figure 3.3: High-level overview of SymDIVINE architecture. Nested boxes correspond to interfaces and their concrete implementations.

- exploration algorithms.

3.3.1 LLVM Interpreter

The interpreter in SymDIVINE operates between input LLVM bitcode and a multi-state space exploration algorithm. It acts as an abstraction layer that provides explicit program representation in the form of a multi-state space graph instead of the implicit one to the exploration algorithms. In this subsection we provide a general overview of the interpreter operation and we describe parts interacting with data store in detail, as understanding the interaction is essential to our work.

The interpreter is represented by the `Evaluator` class in the source code and provides a similar interface to the state generators of explicit-state model checking tools like DIVINE. After initialization of the interpreter with an input LLVM bitcode, it provides a reference to so-called working multi-state (we will refer to it as the "working copy") and provides several functions that can be used to modify this working copy. Exploration algorithm then can then write a multi-state to the working copy and use the interface of the interpreter to modify it and examine it. The following functions are available:

- `initial` function constructs an initial multi-state in the working copy – a multi-state of a program just after the start of the main thread and the call of the `main` function.
- `advance` function sequentially constructs all possible successors of the working copy. The caller is notified about newly produced successor by a callback function.

- `is_empty` function returns true if the set of possible memory valuations of the working copy is empty.
- `is_error` function returns true if there is a violated assertion or a memory corruption (e.g. access beyond array boundaries) in the working copy.
- `push_prop_guard` function filters possible memory valuations of the working copy using a given predicate.

This interface allows for easy mimicking existing explicit-state model checking algorithms like simple reachability or automata-based LTL model checking. A multi-state corresponds to a set of explicit-states, as we mentioned in the general tool overview, so the generator-related operations in these algorithms can be up to exponentially faster compared to their explicit-state version.

To allow the interpreter to operate on top of a user-supplied implementation of a multi-state (here and in a source code referred to as *data store*), several assumptions about the state representation are made:

- control flow location is represented explicitly,
- memory mapping layer (MML) is provided,
- set of functions for data manipulation is provided.

We will now discuss each of these assumptions in detail.

The interpreter expects the control flow location in a straightforward form following the instruction identification in an LLVM bitcode. Each instruction in an LLVM bitcode can be uniquely identified by a triplet $(f_{idx}, bb_{idx}, i_{idx})$, where f_{idx}, bb_{idx} and i_{idx} are indices of function in the LLVM bitcode, a basic block in the function and the instruction in a basic block, respectively. As SymDIVINE supports multi-threaded programs, a control flow location is kept for each thread. There is a unique integral identifier assigned to each thread upon its execution. To represent control flow state of a multi-thread program with function calls, SymDIVINE keeps a stack of instruction identifiers for every thread.

To interpret the verified program, a unique identification of program's variables also needs to be established. Identification similar to the way as instructions are identified is not sufficient, because recursion and nested function calls might occur and it is necessary to distinguish variables in different calls of the same function. As SymDIVINE does not support dynamic memory allocation, inspiration for variable identification was taken from the classical call stack. When a new block of memory is allocated in an LLVM program (function is called or an `alloca` instruction is interpreted), a new memory segment is created and assigned to the operation (function call or the `alloca` instruction). Each automatic variable in the function body or an element of array (in case of the `alloca` instruction) is then assigned an index (in source code and in further text referred to as *offset*) in this segment. Every instance of a live variable can then be identified by its segment and offset. Note that the first segment is reserved for global variables.

Variable naming using segment and offset is required by the interpreter as this fixed naming allows straightforward implementation of strongly typed pointers, arrays and structures. Pointers are simply implemented as a pair containing a segment and an offset. An array of size n is represented as n independent variables. Since the arrays are strongly typed (the `bitcast` instruction is not supported by SymDIVINE), pointer arithmetic for arrays can be implemented as offset manipulation. Similarly, structures are implemented as several independent variables. Since there are no requirements for the offset numbering, memory layout layer, implemented by a data store, allows the interpreter to map variables from currently executed function to a correct offset. To see example of a memory layout layer, follow Section 3.4.

Since the set of possible data valuations in a multi-state can be expressed in many ways, the interpreter relies on the interface provided by the data store. This interface is designed in such a way, that the effect of every LLVM instruction and every intrinsic function definition on a given multi-state can be expressed as a sequence of function application from the interface. The interface is further described in Section 3.3.2.

Given this set up, the implementation of the `advance` function for successor generation is straightforward. In theory, SymDIVINE produces successors by an exhaustive enumeration – every possible thread interleaving is emitted. This approach would cause a state-space explosion well-known from explicit-state model checkers that operate on top of LLVM like DIVINE. To at least partially alleviate the state-space explosion, the interpreter involves the so-called τ-reduction[23]. In practice the interpreter does not emit every direct successor, but keeps traversing the multi-state space graph until a visible action is performed (`load` or `store` on a globally visible variable is interpreted or a safety property is violated). It effectively squashes the effect of multiple instructions with no visible action to single transition and thus removes unnecessary thread interleavings and produces smaller multi-state spaces equivalent with the original one (for safety properties and LTL formulae with no next operator). For details of implementation we kindly refer to [13].

To illustrate the interpreter operation, we present Figure 3.4 and Figure 3.5. The first figure shows an example of a simple C code and the corresponding LLVM bitcode. The second figure shows a multi-state space graph that is produced by the interpreter if the bitcode from Figure 3.4 is supplied as the input.

```
volatile int a;
int main() {
    a = __VERIFIER_nondet_int();
    while(1) {
        int b = __VERIFIER_nondet_int();
        if (b < a) { a = b; }
        if (a == 42) { break; }
    }
    return 0;
}
```

```
@a = common global i32 0, align 4
; Function Attrs: nounwind uwtable
define i32 @main() #0 {
  %1 = tail call i32 (...)* @__VERIFIER_nondet_int() #2
  store volatile i32 %1, i32* @a, align 4, !tbaa !1
  br label %2

; <label>:2                                       ; preds = %7, %0
  %3 = tail call i32 (...)* @__VERIFIER_nondet_int() #2
  %4 = load volatile i32* @a, align 4, !tbaa !1
  %5 = icmp slt i32 %3, %4
  br i1 %5, label %6, label %7

; <label>:6                                       ; preds = %2
  store volatile i32 %3, i32* @a, align 4, !tbaa !1
  br label %7

; <label>:7                                       ; preds = %6, %2
  %8 = load volatile i32* @a, align 4, !tbaa !1
  %9 = icmp eq i32 %8, 42
  br i1 %9, label %10, label %2

; <label>:10                                      ; preds = %7
  ret i32 0
}
```

Figure 3.4: Example of a very simple C code and the corresponding LLVM bitcode obtained as a result of a compilation with Clang with O2 optimizations. The produced bitcode should be simple enough to be understandable even without a deep knowledge of LLVM. Note that variable `a` was marked as `volatile` and thus the compiler cannot optimize out any load or store operations to/from this variable. See Figure 3.5 for the corresponding multi-state space.

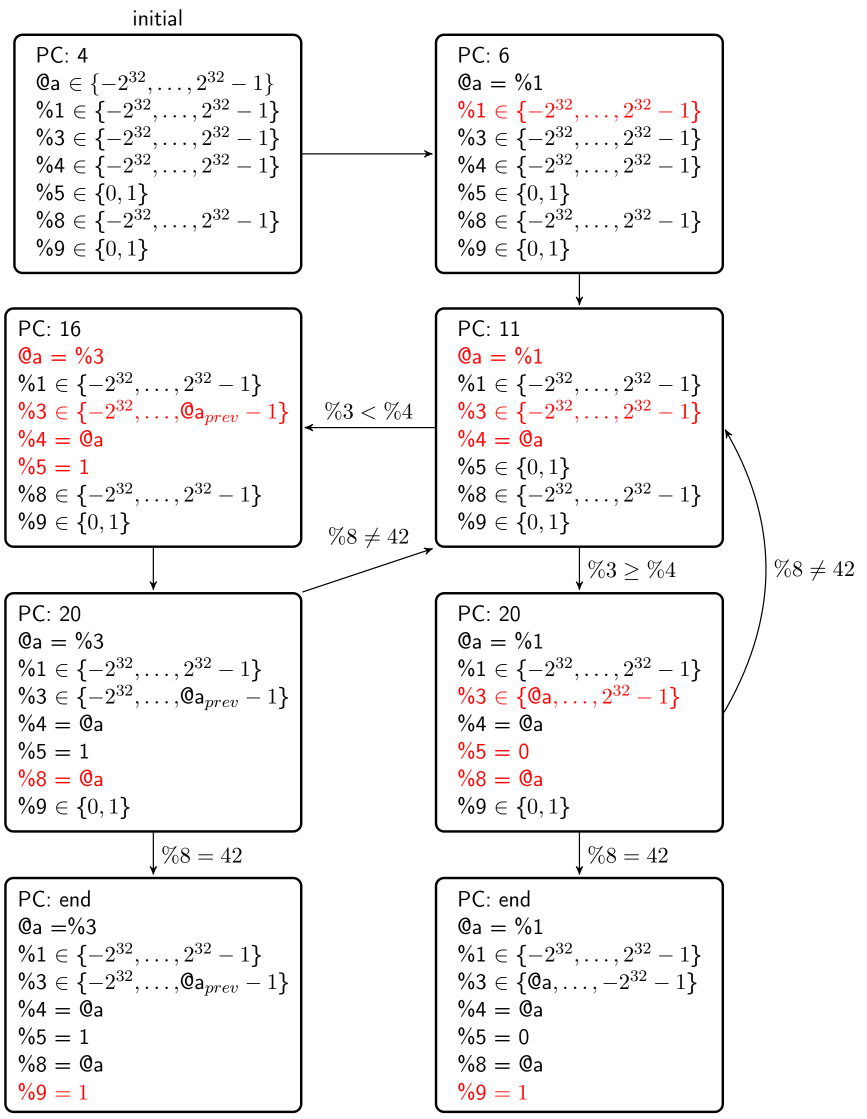
initial
PC: 4
@a $\in \{-2^{32}, \dots, 2^{32}-1\}$
%1 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%3 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%4 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%5 $\in \{0, 1\}$
%8 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%9 $\in \{0, 1\}$
PC: 6
@a = %1
%1 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%3 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%4 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%5 $\in \{0, 1\}$
%8 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%9 $\in \{0, 1\}$
PC: 16
@a = %3
%1 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%3 $\in \{-2^{32}, \dots,$ @a$_{prev} - 1\}$
%4 = @a
%5 = 1
%8 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%9 $\in \{0, 1\}$
$\%3 < \%4$
PC: 11
@a = %1
%1 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%3 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%4 = @a
%5 $\in \{0, 1\}$
%8 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%9 $\in \{0, 1\}$
$\%8 \neq 42$
$\%3 \geq \%4$
$\%8 \neq 42$
PC: 20
@a = %3
%1 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%3 $\in \{-2^{32}, \dots,$ @a$_{prev} - 1\}$
%4 = @a
%5 = 1
%8 = @a
%9 $\in \{0, 1\}$
PC: 20
@a = %1
%1 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%3 $\in \{$@a$, \dots, 2^{32}-1\}$
%4 = @a
%5 = 0
%8 = @a
%9 $\in \{0, 1\}$
$\%8 = 42$
$\%8 = 42$
PC: end
@a =%3
%1 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%3 $\in \{-2^{32}, \dots,$ @a$_{prev} - 1\}$
%4 = @a
%5 = 1
%8 = @a
%9 = 1
PC: end
@a = %1
%1 $\in \{-2^{32}, \dots, 2^{32}-1\}$
%3 $\in \{$@a$, \dots, -2^{32}-1\}$
%4 = @a
%5 = 0
%8 = @a
%9 = 1

Figure 3.5: Multi-state space corresponding to the code from Figure 3.4. As there are no nested function calls, we used simple naming according to variable and register names in the LLVM bitcode. Program counter (PC) is expressed as a line number of instructions that is going to be interpreted next to make the scheme easier to read. Note the τ-reduction in action, where multiple globally invisible actions are squashed together. To make the schematic even more easy to read, we highlighted fields that have been modified by a transition and we also added labels to edges, to make clear which action caused a given transition. If multiple threads were involved, all possible context switches would occur on every transition. Also, please note that to express the valuations set, it was necessary to refer to the value of `a` from previous state. We marked it as a_{prev}.

3.3.2 Data Store

In this subsection we describe an interface of a data store that is used by the LLVM interpreter to analyse and transform multi-states. To see an example of a possible implementation of this interface, please see Section 3.4, where we provide in-depth description of SMT Store. Each data store keeps the explicit part of a state and the symbolic (data) part. We omit description of the explicit part of state as its implementation is trivial. The interface can be split into following categories: memory mapping layer, transformations and analysis.

To present the interface formally, we define a set of possible memory valuations as a function $v : V \to 2^B$, where V is a finite set of program variables and B is a set of all bit-vectors. v also follows that for all $y \in v(x), x \in V$ the bit-width of y matches the bit width of x declared in the LLVM bitcode.

The memory mapping layer is invoked when the interpreter needs to allocate new memory or dereference a register or a pointer. The following functions are required:

- `add_segment(bws)` function – given a thread identifier and a list of bit widths `bws`, constructs a new stack segment for these variables and returns its identifier. Formally speaking, the set of variables V is extended, $v(x)$ for newly-added x is undefined.

- `erase_segment(id)` function – erases segment and guarantees that valuation of variables from other segments is not changed. Formally, the set of variables V is reduced. Value of $v(x)$ for all x that were not in the removed segment also stays the same.

- `deref(tid, id)` – given a thread identifier and a register identifier from LLVM bitcode, it returns an identifier of a variable in the form of segment and offset. If the identifier was a pointer, it returns an identifier to a location pointed to by that pointer. Only values from the global scope or the currently called function in the given thread are allowed as arguments.

Transformation functions are invoked when the interpreter needs to perform an arithmetic operation or store value to a memory. The following functions are required:

- `implement_{op}(a, b, c)` – set of functions that, given three memory locations obtained by call to MML, implement a given binary operation (arithmetic, bitwise, etc.) using `a` and `b` as arguments and storing the result to `c`. Formally, `implement_{op}` changes v to v' such that $v'(x) = \{x \texttt{ op } y \mid x \in v(\texttt{a}), y \in (\texttt{b})\}$ if $x = \texttt{c}$, otherwise $v'(x) = v(x)$.

- `implement_input(a)` – stores a non-deterministic value to given memory location. Formally, `implement_input` changes v to v' such that $v'(x) = \{b \mid b$ is a bit-vector of bit-with corresponding to bit-width of $x\}$ if $x =$ `a`, otherwise $v'(x) = v(x)$.

- `prune_{op}(a, b)` – given a simple relation operator (grater, smaller that, equal to, etc.) and two memory locations, it removes memory valuations in which the relation does not hold.

- `store(r, p)` and `load(r, p)` – given a register and a pointer[5], it either stores a value from register to the memory pointed to by the pointer or loads a value to a register from memory pointed to by the pointer. Formally, `store` changes v to v' such that $v'(x) = v(r)$ if `p` points to x, otherwise $v'(x) = v(x)$. `load` operation is defined symmetrically.

The last category are analysis functions used mainly by exploration algorithms to construct a set of known multi-states and produce a product with an automaton:

- `empty(a)` – returns true if the set of possible valuations of `a` is empty. Formally, returns true if and only if there is an x such that $v_a(x) = \emptyset$.

- `equal(A, B)` – given two multi-states, returns true if the program counter of both states is the same and the sets of possible valuations are the same. Formally, returns $v_A = v_B$. Note that the sets of program variables V_A and V_B are the same, as SymDIVINE does not support heap allocation and the program counters are equal. Also note that there might be representations, whose equality cannot be checked purely by syntactic or memory equality, as we show in an example Section 3.4.

- `get_explicit_part` – returns an encoded explicitly represented part of the multi-state in the form of a binary blob. If two multi-states are equal to each other, both blobs from the multi-states have to be the same.

- `less_than` – if there is an ordering defined on the multi-state representation, this function can be provided (and thus an algorithm can use a tree set to represent a set of multi-states)

There were several implementations of data store developed. A short summary of them follows. Note that not all of them are distributed in the current release of SymDIVINE as they were replaced by a more efficient one, or their development and support was discontinued.

[5]Note that `store` can also take a constant instead of aregister. As it is a technical detail, we omit this variant in the text.

- Explicit store – represents only a single possible memory valuation, not a set of valuations. Usage of this store "degrades" SymDIVINE to a purely explicit-state model checker. This store is used for implementation of the explication optimization to reduce number of multi-state equality test during state space exploration. As this optimization is not important for our thesis, we kindly refer to cite{Havel2014thesis} for further detail.

- BDD store – BDDs are used to represent a set of possible memory valuations. There are algorithms for computing binary arithmetic and logic operations for BDD [9], so the implementation of a store is straightforward – for every program's variable there is a single BDD. An equality check of two BDD s is a cheap operation, as they feature canonical representation. However, the construction of a BDD for arithmetic operations (e.g. multiplication) is quite expensive. This kind of store failed to verify even small examples due to the high complexity [13]. Thus, development of this store was discontinued

- SMT store – uses an SMT formula to represent a set of possible memory valuations. To decide whether two representations describe the same set of valuations, an SMT solver for quantified bit-vector theory is used. For further description of this store, follow Section 3.4, where we describe this store in detail. The store is used as the primary one in the current release of SymDIVINE.

- Empty store – does not represent any memory valuations and only a collects sequence of transformations applied to the store. This is not useful for any verification technique, however, it can be used to translate an LLVM bitcode into different kinds of formalisms. See next section where we describe this process in more detail.

3.3.3 Exploration Algorithms

On top of the LLVM interpreter and a data store it is easy to implement an algorithm for state space exploration. The algorithm is usually the only thing user interacts with. Taken all inputs from the user (LLVM bitcode, property, exploration strategy etc.), it usually instantiates an interpreter, asks for an initial state and, using the `advance` function of the interpreter, it builds a set of known multi-states or even the full multi-state space graph.

As a multi-state is required to provide a procedure for equality of two multi-states, it is possible to represent a set of multi-states. A set representation using only a an equal operation would not scale well to real-world program sizes for obvious reasons. Note that a traditional hash-set used in explicit-state model checker cannot be used as there can be a multi-state representation that doesn't have a canonical form (e.g. SMT store). Thus, SymDIVINE tries to benefit from having an explicit control flow is mandatory for every data store implementation. A typical set of multi-states is implemented as follows: there is a hash map containing a list of symbolic parts of the multi-states for every explicit part of the multi-states. When a new element is inserted to the set, list of the symbolic parts corresponding to the explicit part is recalled and then every symbolic part from the list is tested for equality. If an equal symbolic part is found, procedure ends; otherwise new symbolic part is put at the end of the list. This optimization significantly reduces the number of calls to the equality procedure, however, it is still possible to obtain a significant number of symbolic parts per control flow location. If a multi-state representation allows to implement the `less_than` procedure, it is possible to replace a linear search by binary search and thus further optimize the set representation.

Using various combinations of algorithms and data stores, SymDIVINE can serve as a multi-purpose tool. During development of SymDIVINE, experiments with the following combinations were performed:

- SMT or BDD store combined with an algorithm for reachability. This combination produces a model-checker for safety properties that can handle input values. This approach was originally introduced in [13].
- SMT or BDD store combined with a standard algorithm for automata-based LTL model-checking. During the verification, negation of specification LTL formula is converted to a Büchi automaton and during the successors generation procedure a product with the automaton is produced. A test for atomic propositions that can refer to global variables of the program is implemented using the `prune` operation of the store. It filters-out memory valuations that violate the atomic proposition – see Figure 3.6. This approach was originally implemented in [6] and further improved in [20].
- Explicit store combined with reachability or an LTL algorithm on input programs with no non-deterministic input produces a standard explicit-state model checker.
- SMT or BDD store combined with simple exploration without tracking the set of known multi-states yields in a symbolic execution.
- Empty store in combination with reachability can be used to convert LLVM bitcode to an artificial modelling language. Thus, tool like nuXmv [11], that does not support LLVM as an input formalism, can be used to verify properties of an LLVM bitcode. When running the reachability, the empty store produces one state per each reachable control flow location and collects sequence of transformations applied to the state and transition guard – constructs a non-deterministic guarded transition system. After exploration, this transition system is translated to desired modelling language. This usage of SymDIVINE was introduced in [6].

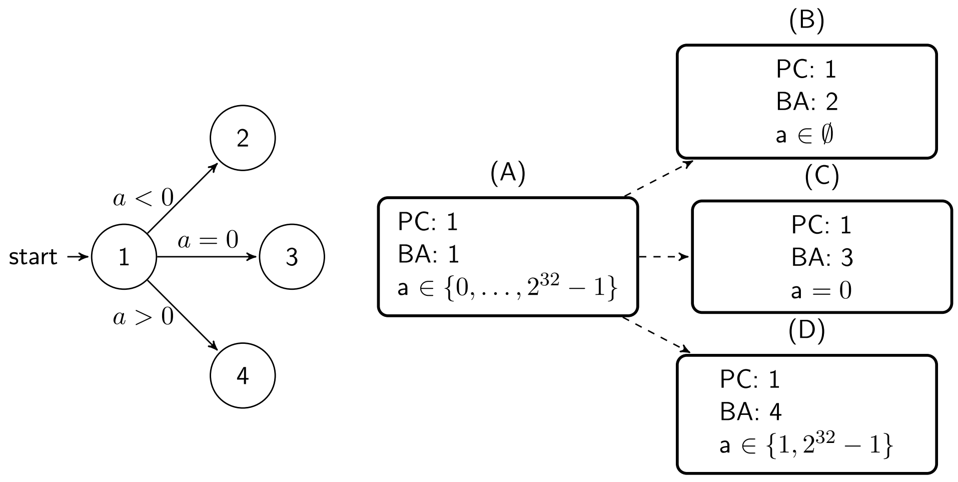

Figure 3.6: Illustration of a multi-state space generation in an automata based LTL model checking. The LTL exploration algorithm takes a Büchi automaton (a subset of such a automaton is shown in the left). The generation procedure works as follows: state that is being explored is loaded into the interpreter and a new successor (state A) is produced. The state contains a program counter, a Büchi automaton state and a set of possible data valuations. The product (states B, C and D) is generated by taking all possible transitions of the automaton from a given state by pruning the set of valuations. All non-empty states (C, D) are then emitted as successors of the explored state.

3.4 SMT Store

In this section we closely look at the implementation of SMT Store, as understanding of its internals is essential for our work. First, we describe the store from a theoretical point of view and then we closely look at the actual implementation, which features several optimizations and thus slightly differs from the theoretical model.

3.4.1 Theoretical Model

SMT store uses a representation described in [2]. A quantifier-free first-order bit-vector SMT formula φ called *path condition* is used for description of the possible memory valuations set defined in Section 3.3.2. The set of program variables V (defined in Section 3.3.2) can be mapped to the set of the free variables in the path condition (in a way we describe later in the text). Program variables can be sequentially assigned different values during a single program run. To distinguish different values assigned to a variable, a so-called *variable's generation* is used. For each assignment to a program variable, a new variable in formula is created (with incremented generation number). Thus, the set of variables in formula is a subset of $V \times \mathbb{N}$ and we can denote a formula variable as a pair (v, g), were v is a program variable from V and g is its generation. The mapping from formula variables to program variables is easy – every program variable v maps to formula variable (v, g) with the maximum generation g. The set of all models of path condition defines the valuation function v.

Implementing the data store interface using path condition is performed in the following way (as described in [13]): when we refer to a formula variable, we refer to the latest generation. We also assume implicit mapping of program variables to formula variables (when we use program variable in formula, we assume it is translated to formula variable with maximum generation).

- `add_segment(bws)` and `erase_segment(id)` functions do not modify the path condition, only meta information used for implementation is modified.

- `implement_{op}(a, b, c)` on state with a path condition φ leads to a new path condition: $\varphi \wedge ((\texttt{c}, g+1) = \texttt{a op b})$.

- `implement_input(a)` increases generation of `a` and does not modify path condition, as a variable with no constrains models a non-deterministic value.

- `prune_{op}(a, b)` on a state with a path condition φ leads to a new path condition: $\varphi \wedge (\texttt{a op b})$.

- `store(r, p)` and `load(r, p)` on a state with w path condition φ leads to a new path condition: $\varphi \wedge ((\texttt{p}, g+1) = \texttt{r})$ in case of the `store` instruction, `load` instruction is defined in a symmetrical manner.

- `empty` returns true if and only if the path condition is not satisfiable. Satisfiability is decided using an SMT solver.

- `equal(A, B)` returns true if and only if the path condition φ of `A` and path condition ψ of `B` represent the same set of possible valuations. There is no canonical form of SMT formulae, thus two different formulae can describe the same set and it is not possible to decide equality using purely simple syntactic equality. Note that there also cannot be any `less_than` function implemented for this data store. A quantified bit-vector SMT query is made to a solver in order to decide the equality. `equal(A, B)`, returns true if and only if:

$$\neg\, \text{notsubseteq}(\texttt{A}, \texttt{B}) \wedge \neg\, \text{notsubseteq}(\texttt{B}, \texttt{A})$$

is satisfiable, where notsubseteq(`B`, `A`) is a short-cut for:

$$\exists b_0, \dots, b_n.\, \psi \wedge \forall a_0, \dots, a_n.\, \varphi \implies \Big(\bigvee (a_i \neq b_i)\Big)$$

where $a_0, \dots, a_n$ denotes the program variables in `A` and $b_0, \dots, b_n$ denotes the program variables in `B`. Intuitively, notsubseteq looks for a valuation in `A` that is not present in `B`.

Given the implementation of the operations, we can now easily illustrate the need for different variable generations. Consider the following example of an LLVM bitcode:

```
store i32 5, i32* %a, align 4
%2 = load i32* %a, align 4
store i32 %2, i32* %b, align 4
store i32 42, i32* %a, align 4
```

This piece of bitcode stores constant 5 to `%a`, then assigns the value of `%a` to `%b`. The last operation stores constant 42 to `%a`. With no generations, the last store operation would change both values of `%a` and `%b`, so it is necessary to keep track of each variable's history.

3.4.2 Implementation

To achieve a better performance of SMT store, several optimization to the purely theoretical approach are made. In this section we first describe the implementation of meta information that is needed to correctly build a path condition. Then we describe the optimization of path condition building and the evaluations of the `empty` and `equal` operations.

SymDIVINE uses the Z3 SMT solver [19] to decide the satisfiability of both quantifier-free and quantified SMT queries. In order to allow easy usage of other solvers, SMT store relies on the internal formulae representation. The path condition is stored in this internal representation. When an SMT query is needed, the internal representation is translated to solver's specific format.

To correctly and effectively build a path condition, preservation of the set of program variables, their mapping to formula variables and their generations is needed. Also, as we described in section Section 3.3.1, the interpreter requires that data store provides MML – thus a mapping among variables in LLVM bitcode and program variables (where multiple calls of the same function are allowed) needs to be established. We remind that program variables are required to be identified by a segment and an offset.

To perform these mappings, SMT store takes advantage of the required segment-offset program variable identification and thus keeps a set of segments with variables and a mapping of call stack frames to these segments. Note that this mapping is not canonical – different thread interleaving can lead to a different mapping. Each segment contains a list of variables (information about the highest generation, bit-width, etc.). See Figure 3.7 for an illustration of this mapping.

As the set of segments is usually quite small, frequently accessed and changed, an implementation using a dynamic-sized array and a free-list was chosen. Thus the segment identifiers are re-used and the same segment in two different states can be mapped to a different stack frame.

On top of this set-up, implementation of the data store interface is straightforward. Note that during the `equal` operation a set of variables pairs that are compared during the notsubsetq operation needs to be computed, as the segments mapping is not canonical and we cannot directly compare variables from the same segment number in different states.

This naive implementation does not scale well, as path condition grows quickly (LLVM bitcode uses a considerable number of registers, because LLVM is a static single assignment language) and an enormous number of expensive quantified SMT queries is performed. Thus, SMT store uses the following optimizations:

- Unused variable definitions – the path condition is split into two parts: list of definitions in the form $\mathit{variable} = \mathit{expression}$, and a list of path condition clauses in the form of $\mathit{variable}\, \mathrm{rel}_{\mathrm{op}}\, \mathit{variable}$[6]. Definitions are mainly collected during arithmetic, `store` and `load` instructions. Path condition contains only constraints collected during the `prune` operation. This severance to two independent pieces allows us to easily remove unused definitions without a complicated analysis when the function returns. The return value can be easily expressed through simple syntactic substitution using only function parameters and all variables from the function segment can be removed from the definitions and the path condition clauses. When a formula is needed, a simple conjunction of all definitions and path condition clauses is constructed. This optimization significantly reduces the size of path condition (and thus the size of SMT queries to a solver) and also keeps the set of program segments small.

- Syntactical equivalence checking – as SymDIVINE aims for verification of parallel programs, the same path conditions due to diamond-shapes in the multi-state can be produced. As SMT query to a solver is expensive operation even for simple queries [13], SymDIVINE tries to perform simple syntactic equality test of path conditions before executing this query. Non-trivial number of solver calls can be saved in this way.

- Simplification – Z3 SMT solver offers several simplification strategies that can be applied to a path condition. These strategies can be applied to a

[6]a constant can also figure in such a clause

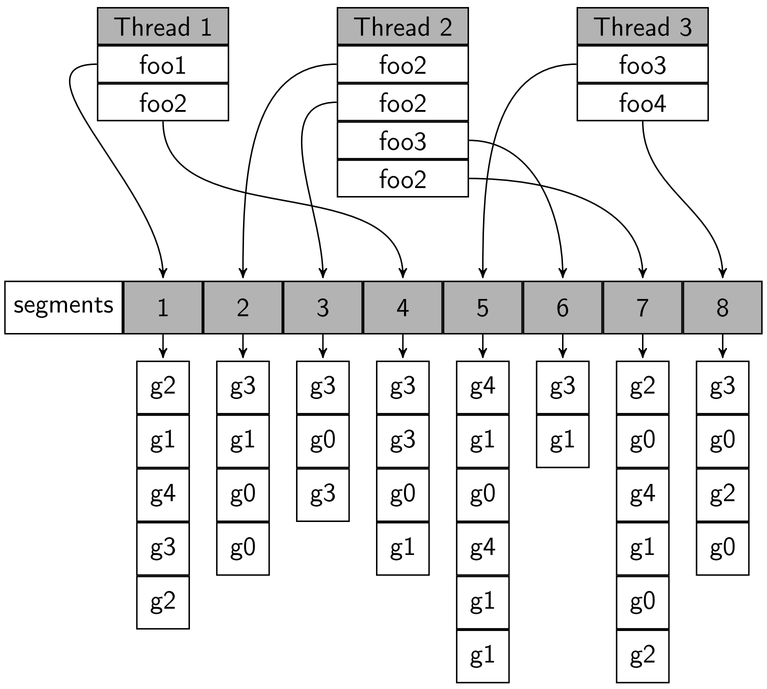

Figure 3.7: Illustration of path condition meta information organisation in SMT store. Each thread has its own call stack. When a function is called, `add_segment` is called on the SMT store and a new segment is created. Each segment points to a list of variables in the given segment that keeps track of currently-highest generation of each variable as shown in the picture. Note that the mapping call stack frames to a segment is not canonical and, depending on thread interleaving, it may vary.

path condition in order to produce smaller or easier forms of equivalent path condition. Simplification is applied when a new formula with a large number of variables is produced.

Chapter 4

SMT Queries Caching

Trading off time for space is a basic approach to improving the time efficiency and scalability of software tools. Storing the intermediate results of intermediate can be significantly less time-consuming than re-computation and therefore cause a speed-up of a tool.

In this chapter we present our motivation for proposing and implementing several SMT queries caching techniques for SMT data store in SymDIVINE. We provide a quick overview over existing caching solutions and their suitability for our case. We then propose and describe in detail two approaches that we find interesting – a naive one, which leads to another optimization in SymDIVINE, and a dependency-based one, which brought up a noticeable speed-up. In the following Chapter 5, we present experimental evaluation of our implementation of these techniques.

4.1 Motivation

Scalability is an important factor of verification tools, which aims for verification of real-world sized code. During various experiments with SymDIVINE, we noticed that most of the verification time is spent on Z3 SMT solver calls. We analysed SymDIVINE using time measurements and Callgrind tool [27] on verification task from SV-COMP concurrency set[7]. SMT data store in SymDIVINE performs two kinds of SMT queries – a quantified query for decision of multi-state equality and a quantifier-free query for emptiness check (both described in Section 3.4). In average, roughly 70 % of the time is spent on quantified SMT solver calls and 10 % of the time is spent on quantifier free queries.

There are two reasons why we consider caching of queries effective. First, SymDIVINE constructs a path condition in a similar manner as symbolic execution. Thus, a constantly growing formula sharing a common prefix is constructed. This growth is also supported by the fact that LLVM is a single static assignment language and as such uses enormous number of registers.

The multi-state emptiness check is performed in the same manner as in the symbolic execution.

Second, SymDIVINE focuses on multi-threaded programs and various thread interleavings can cause so-called diamond-shapes in the multi-state space. This means that two paths in a multi-state space join to the same state and an equality check has to be performed. More of the same or similar diamond-shapes can occur multiple times in different parts of a multi-state space. Syntactic equality optimization works in some cases. However, there might be a diamond-shape which can be resolved only by using an SMT query. When such a diamond-shape appears multiple times, the same queries to an SMT solver needs to be issued.

Caching of both empty and equal queries can bring a speed-up. There are various caching techniques to speed-up SMT queries that might work for the emptiness check (a batch of quantifier-free queries sharing a common prefix), however, we are not aware of a technique which could help us in case of an equality check (a batch of quantified SMT queries which includes a sub-formula with a growing common prefix). We also see bigger potential in the equal query, as it is computationally more demanding. The traditional techniques do not work, as the quantifier in the query makes slicing it into cacheable parts computationally hard or even impossible in some cases. As our experience shows, making a query to the Z3 brings a non-trivial overhead even for easy queries and therefore we would like to avoid it (the same phenomena was observed in [13]), if possible.

We think bringing a knowledge of our setting (fixed format of the equal query and its semantics or path condition origin) can help us to face the issue and design an effective caching technique.

4.2 Classical Approaches In Other Tools

There are no resources we are aware of which in detail describe all caching optimizations implemented in Z3. Only a brief overview can be found in [19]. However, from this overview and our shallow knowledge of Z3 source code, we assume that, in principle, the caching optimizations work in a similar manner as optimizations that can be found in KLEE [21], PEX [24] (both are tools for symbolic execution) or in GREEN [25] (framework for caching SMT queries during symbolic execution). Z3 also features a cache for the built-in SAT solver, on top of which the SMT solver operates.

In principle, two main approaches to caching can be found – constraints caching and unsatisfiable cores caching. Both of these approaches are adapted for purposes of symbolic execution and are designed to handle a batch of quantifier-free SMT formulae with common prefixes.

Constraints caching takes advantage of the way a path condition in symbolic execution is built. As the symbolic execution collects new constraints, new conjuncts are added to the formula. Thus, the queries follow the form $\varphi \wedge \psi$, where φ denotes the known part of the path condition (that was already issued as a query) and ψ denotes the part of the formula with new conjuncts. If φ is not satisfiable according to a cached result, the whole query cannot be satisfiable. Otherwise φ and ψ are syntactically analysed and only conjuncts from φ that share a data dependency with ψ are taken. Satisfiability of this smaller formula is then decided. To effectively select only the necessary conjuncts of φ, GREEN builds a tree structure over existing parts of the path condition [25]. Also before deciding satisfiability, parts of the formulae are canonized to increase the chance of a cache hit.

Unsatisfiable cores caching can be seen as an extension of the previous techniques. When an unsatisfiable query is issued, the unsatisfiable core is computed and transformed into a pattern. When a new query is processed, it is first checked for a presence of unsatisfiable patterns which have occurred so far.

These approaches work well for quantifier-free queries which are produced during symbolic execution. However, from our experience and experiments performed with Z3, we assume that these caching optimizations are not applied to quantified queries at all or do not have any noticeable effect. We are also not aware of any work that would specialize on caching of quantified formulae.

4.3 Naive Approach

We began our experiments with a naive approach – caching of whole query for multi-state equality. This naive approach can be seen as an extension of syntactic equality optimization, which was described in Section 3.4.2. Syntactic equality can eliminate a query to an SMT solver in case of diamond shapes that result in the syntactically same path conditions. If this diamond-shape results in a syntactically different path conditions, SMT solver has to be called. If such a diamond appears in multi-state space again, the same query is performed. If naive caching is present, the second query to an SMT solver can be eliminated.

We have implemented this naive approach in SymDIVINE version from [6]. The naive caching is implemented using a hash-map from an SMT query to a result of such query. In the original implementation, `equal` query was directly constructed in the SMT solver native format. We have taken an advantage in form of the fixed query format, as we wanted to keep the caching process independent of an SMT solver (queries are not constructed using formulae representation of SymDIVINE, but directly in the target SMT solver format) and also to minimize memory footprint. Only the essential parts to uniquely identify a query are kept in the table – list of variables pairs to compare and path condition clauses. No other unnecessary parts of the syntactic tree are kept. Just before the real SMT query is constructed, this footprint of query is constructed and checked for presence in the cache. If so, cached result is returned. Otherwise real query is constructed, executed and the result is inserted into the cache.

We have evaluated naive approach using a subset of SV-COMP benchmarks (mainly concurrency and bit-vector tasks) and a set of LTL benchmarks, that have been used in evaluation of SymDIVINE in [6]. Naive caching saved only about 2 % in the reachability tasks, however up to 65 % of the queries were cached in the LTL benchmarks.

This result made us revisit the implementation of the LTL algorithm in SymDIVINE. SymDIVINE uses nested DFS with iterative deepening, as bugs in software are usually shallow and occur e.g. during the first few iterations of a cycle in a program. If SymDIVINE finds a cycle that needs to be unrolled in a verified program, the classical DFS approach will first fully unroll the cycle and then will search the other parts of the multi-state space. Iterative deepening can prevent this behaviour and thus speed up verification of erroneous programs. However, verification of programs with no bug takes longer. The inspiration for the implementation was taken from DIVINE, which also features iterative deepening DFS. DIVINE does not keep the state space graph and during every iteration with increased depth it regenerates the state-space from scratch. However generation of multi-state space is computationally more demanding compared to generation of explicit-state space; the overhead caused by re-generation of the multi-state space is not negligible, as queries to an SMT solver are involved. The re-generation of the multi-state space caused enormous hit-rate and thus brought a significant speed-up. Note that similar effect was not observed on reachability, as it uses a BFS based approach.

As the original LTL algorithm did not a feature user-friendly way to pass an LTL property, we decided to implement a new version which would keep the whole multi-state space including the transitions between states and would bring the user-friendliness. Keeping the transitions between the states caused a slightly bigger speed up than naive caching in the original version. Also the hit-rate of naive cache was reduced to similar levels as in case of reachability verification tasks. Memory overhead of this solution is negligible taking into account the size of an average multi-state.

4.4 Dependency-Based Caching

In this section we introduce our approach for caching equal queries in SMT data store in SymDIVINE. Our approach shares similar ideas as constraints caching; however, it uses an additional information about the structure of the query that we can take advantage of.

Let us briefly revisit the structure of equal query in SymDIVINE. The equal query for states A and B is split into two separate tasks – test if A is not a subset of B or if B is not a subset of A. Each notsubseteq test performs the following query to an SMT solver:

$$\exists b_0, \dots, b_n.\, \psi \wedge \forall a_0, \dots, a_n.\, \varphi \implies \Big(\bigvee (a_i \neq b_i)\Big)$$

where φ is a path condition of A, ψ is a path condition of B and $a_0, \dots, a_n$; $b_0, \dots, b_n$ denotes variables from A, B respectively.

The unsatisfaible cores caching has no effect on our query, as SymDIVINE ensures that φ and ψ are both satisfiable independently (empty multi-states are never checked for equality). To apply constraints caching, detecting the new parts of formula (compared to already seen formulas) is needed. Here we face the problem that we cannot easily split the formula in a similar manner into conjuncts due to the presence of a universal quantifier, which captures part of the formula and also the implication in the quantified part.

These issues could be solved by detecting the formula growth on ψ, φ and the sets of variables. Then a dependencies across these part could be computed and a new, smaller query could be produced. However, we see this as complicated and rather computationally challenging. Instead, we take another point of view by using the semantics of the query and the possibility to change the equality procedure to suit the needs of caching.

In our approach we represent multi-state data as a multiple independent sets of valuations instead of a single one. We define two sets of valuations as independent when every change on one of them leaves the second one unchanged. In our case of valuations representation using path conditions, each set is defined by a single path condition. We can simplify the requirements for independence to following: two path conditions are independent if they share no common variable. This requirement is stronger, however, from the practical point of view, they are the same as our path conditions do not contain unused variables. This set-up can be also seen as splitting a program's multi-state into several smaller mutually independent states. We will refer to these as sub-states. See Figure 4.1 for illustration of dividing a multi-state to a set of sub-states.

This set-up reflects commonly seen situation in multi-states of a verified program. During nested function calls, many registers of the programs are left unmodified and they have no effect on current multi-state transformation the during interpretation of an LLVM bit-code. Thus, we can isolate these registers to (even multiple) independent states, which are not modified during current transformation. Another situation can occur during verification of multi-threaded program. It is possible to have two threads which do not communicate (share no memory). Using sub-states, each thread can operate in its own sub-state and advance of one thread does not modify sub-state of the other one.

A sub-state can be in the context of a single multi-state uniquely identified by set of variables it contains – we call this set a *sub-state label*. We say that *two sub-states from different multi-states match* if and only if they have the

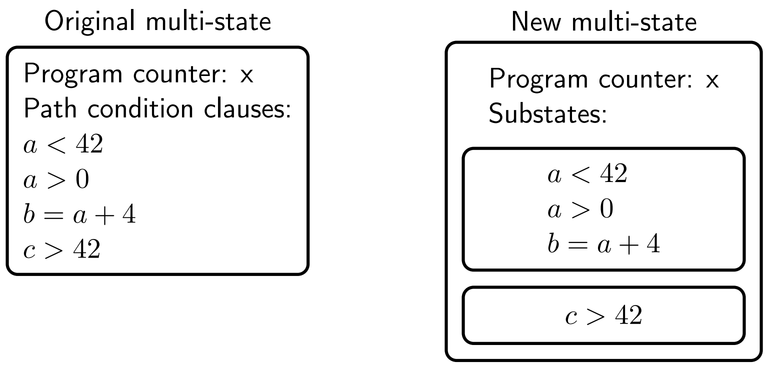

Figure 4.1: Illustration of a new representation of a multi-state. Instead of keeping one path condition, multiple mutually independent path conditions are kept – so-called sub-states. This set-up allows more effective implementation of equal query when using caching.

same label. We say *two multi-states A and B match* if and only if there is a matching sub-state in B for every sub-state in A. A multi-state can be divided into sub-states in many ways. To decide whether two multi-states are equal, these two multi-states have to match. Provided this set-up, we can say states A and B are equal if and only if every sub-state from A is equal to its matching state in B. As all sub-states are independent, proof of this statement is trivial.

To be able to decide equality of any two multi-states, we define operations *split a sub-state* and *merge two sub-states* that allow transformation of every two states into a matching form. Merging of sub-states is straightforward, we create a conjunction of their path conditions and make a set union of their labels. Splitting a sub-state into two sub-states is possible if clauses of the original path conditions can be split into two sets of clauses such that they are independent (share no common variable). We call a sub-state *trivial* if it cannot be split any more. Note that every two multi-states (with the same control part) can be transformed into a matching form, as we can in the worst-case scenario merge all sub-states to a single one and thus produce a sub-state equivalent to a multi-state without sub-states.

Our motivation for introduction of sub-states is straightforward – provided the above-mentioned equality procedure and keeping as many trivial sub-states as possible, we produce smaller and simpler queries to an SMT solver. We also expect a high cache hit-rate, as, in real-world programs, only very few variables have effect on current transformation of a multi-state.

Compared to performing similar operations directly on the queries produced by SMT store, we can compute the data dependencies on the fly with no significant overhead during path condition generation. Thus, we avoid the overhead of building a large query for an SMT solver followed by its analysis and slicing. Instead, we can directly produce small queries which can be cached and also take benefits of other optimizations in SymDIVINE, which can work on top of these small queries. We call this approach *dependency-based caching.*

4.5 Implementation of Partial Store

In this section we provide a detailed description of our dependency-based caching implementation in SymDIVINE and make an overview of small differences to the theoretical description provided in the previous section.

We have implemented this caching technique as a new data store – *partial store.* As a non-trivial part of the data store interface is implemented in the same manner as SMT store (e.g. all `implement_{op}` functions), we abstracted them to a new base class. Both SMT and partial store are derived from this class. Thus we needed to provide only segment related functions, `deref`, `load`, `store`, `prune` and `implement_input` functions.

We have implemented a data structure called *dependency group.* This structure represents a sub-state from the previous section. Dependency group keeps its label, a list of path conditions and a list of definitions as it implements the same optimization as SMT data store. It provides interface for performing dependency groups merging and splitting. Several support functions of purely technical character are also implemented.

Instead of path condition and definitions, partial store keeps a set of dependency groups and a mapping from variables to these dependency groups. When a new variable is created, it is not dependent on any other and thus a new dependency group is created for every newly created variable. When a segment with all variables is destroyed, substitution of variable definitions is performed just like in SMT store. As the dependency groups are independent, substitution occurs only in the context of a single group. If the group label is empty after deletion, the group is destroyed.

If a `store` or `prune` operation is issued, variables from an expression or a constraint are collected, their dependency groups are located and merged. Definition or a path condition is then inserted into the group. This implementation keeps the invariant that dependency groups are always independent. When performing a `store` or `prune` operation would violate the invariant, the affected groups are merged.

The other operations are implemented in the same manner as operations in SMT store. The only difference is the corresponding dependency group has to be located first.

Test for state emptiness is performed for each resource group independently and each group caches the result of this check. If path condition is modified, the result in cache is discarded and the check is repeated. This is a small optimization, which was not introduced in the theoretical description. It produces small, in many cases hardly-measurable speed-up.

To perform an equality check, multi-states need to be first converted to a matching form. Not only dependency groups can differ, a variable naming can also be different as the mapping between the call stack and segments is not canonical. To effectively compute which group needs to be merged, we first obtain a list of variable pairs to compare, just like SMT store does. Then we iterate over this list and, using union-find, we build sets of groups which need to be merged according to groups labels. Then a standard equality check from SMT store is performed for each merged group. To cache these calls, the same approach as the naive one is used.

Chapter 5

Results

In this chapter we present an experimental evaluation of our dependency-based caching and we discuss strengths and weaknesses of our approaches. We have evaluated both algorithms (reachability and LTL) from the current version of SymDIVINE with dependency-based caching.

5.1 Benchmark Set and Environment

To evaluate reachability, we have taken a subset of C benchmarks from SV-COMP [7] benchmark suite. Benchmarks from the following subdirectories were taken: bitvector, eca, locks, loops, recursive, ssh-simplified, systemc, pthread, pthread-atomic, pthread-ext, pthread-lit and pthread-wmm. To evaluate LTL algorithm, we have used LTL benchmarks, which have been used in [6] for benchmarking of the first LTL implementation in SymDIVINE.

Our test machine features Intel Core i5-4690 CPU (3.50 GHz) with 16 GB of RAM and runs Arch Linux distribution with 4.4.8-1-lts Linux kernel. SymDIVINE was built in the release configuration using Clang 3.4 and Z3 SMT solver version 4.4.1-1.

Three test files were produced from a single input benchmark file by compiling it into LLVM bit-code with three different levels of optimizations – `O0`, `Os` and `O2`. Note that each LTL benchmark is shipped with its specification in the form of LTL formula. We have tested each LTL benchmark for its specification and negation of the specification.

First of all, we ran SymDIVINE with different configurations (solver time-out, simplification strategies, etc.) for both SMT data store and partial SMT data store to find the optimal configuration for our set of benchmarks. SMT data store performs best with the default setting (no command line flags – advanced simplifications of the path condition and syntactic equality optimizations are enabled). Partial SMT data store performs best with the same setting, however, simplifications of path conditions have to be disabled, as changing the the path condition using simplifications leads to a zero cache hit-rate.

Using the optimal settings mentioned above, we ran SymDIVINE with partial SMT store and caching enabled on the benchmark set with a time-out of 4 minutes for each task. Then, we ran the same set of benchmarks without caching using the original implementation of SMT store and time-out increased to 15 minutes. The time-out was increased in order to see improvements caused by caching, as many benchmarks without caching time-outed and no relevant data cannot be obtained. Simple benchmarks with no equal queries were excluded from the final results as they are not relevant to caching.

We also verified correctness of the implementation of partial SMT store. We implemented a so-called *validity test* in partial SMT store. This validity test keeps 2 multi-states – one represented by SMT store and the other one by partial SMT store. All multi-state manipulations are performed simultaneously on both stores. When an empty or an equal operation is performed, results of partial SMT store are tested against SMT store. The results have to match. We ran all benchmarks mentioned above using this test and not a single mismatch occurred.

5.2 Evaluation

We examined the results of each category independently, to see the effects of caching on different types of input programs. For summary results of our measurements[1], follow Table 5.1. We looked at the verification time and number of queries to an SMT solver. A short evaluation of results for each category is provided below:

bitvector There are many simple benchmarks in this set that contain only the necessary constructions to produce a bug in bit-vector manipulation. Figure 5.1 shows that caching overhead is compensated with its positive effect on small benchmarks and therefore verification time differs only by less than one percent. On the other hand, there are benchmarks like `gcd*`, where caching saved over 50 % of verification time, as there is a sub-formula in the path condition that causes troubles to the Z3 solver. Using caching, this sub-formula is used only once as the result is cached. In summary, almost half of the verification time can be saved using caching.

eca This is the single category that exploited weaknesses of our caching approach as can be seen in Figure 5.2. Benchmarks from this set are generated pieces of code with an enormous number of variables and non-trivial dependencies and therefore the overhead of caching is noticeable. Even compilation of these benchmarks takes an unexpected amount of time (usually a few minutes).

[1]Full measurements can be found in B

locks Almost all benchmarks in this category are simple enough to be directly solved by optimization passes in Clang– optimizations in the compiler are able to simplify the benchmarks up to a single branching. Therefore SymDIVINE produces only 3 multi-states, and thus, the results, which can be seen in Figure 5.3, are not significant.

loops Many benchmarks in this category suffer from the same issue as benchmarks in locks category – many of them can be solved by the compiler itself. However, verification time of complicated benchmarks can be reduced in summary by almost 30 %, see Figure 5.4. This observation is in line with our expectations (equality of intermediate results in program run is evaluated only once when caching is used).

recursive There are no effects of caching in programs with recursion, as can be seen in Figure 5.5. This is due to the fact that multi-states produced in recursion cannot be merged, as their explicit control-flow location differs.

ssh-simplified and systemc Benchmarks from this category are quite large and feature a similar structure to loops benchmarks, therefore caching provides a good performance and can save in average about half of the verification time (in some benchmarks even 75 % of verification time can be saved). These results can be seen in Figure 5.6 and Figure 5.7. Note the significant decrease in number of solver queries.

concurrency We expected the most significant effect of caching in concurrency benchmarks due to the presence of diamond-shapes in multi-state space. Even though almost half of the verification time is saved using caching, there is systemc category, where caching performed slightly better. The reason for such a behaviour is as follows: many equal queries on diamond-shapes can be optimized out by the syntactic equality optimization. This optimization does not apply to other categories, as the other benchmarks are sequential – compare number of equal queries and solver calls in Table 5.1). Therefore there are not as many SMT queries, which can be cached. With disabled syntactic equality optimization, caching saves in summary 95 % of verification time and issue only 2 % of solver queries (1769313 queries without caching compared to 45917 queries with caching), which meets our original expectations. Also, benchmarks in this set do not contain complicated arithmetic, so the queries to an SMT solvers are quite simple and the benefit of caching is not significant as in other categories.

LTL There are two kinds of benchmarks in this set; ones that are very simple and contain almost no arithmetic, and the others, that contain a loop, which has to be fully unrolled and therefore they cannot be verified by SymDIVINE in a reasonable amount of time. An interesting phenomena appeared in these simple benchmarks – all equal queries were decided using only syntactic equality. When a product of a multi-state space and an automaton is generated, transition guards are pushed to the multi-states and diamond shapes are produced. However, pushing transition guards in different order produces a syntactically different path condition and therefore no syntactical equality was detected. In contrast, pushing transition guards to sub-states in different order produces syntactically equal path conditions (as the conjuncts are not interleaved). Even though no queries were made, no speed-up occurred, as can be seen in Figure 3.6.

Overall We observed following behaviour from all our test runs: in summary, caching reduces verification time by 44.8 %. Caching applies mainly to the large benchmarks, where it can save up to 75 % of verification time. If caching does not bring a speed-up, usually only a negligible slow-down in units of percent occurs (except benchmarks from the artificial eca set). The overall positive results are summarized in Figure 5.10. We can conclude that the speed-up produced by dependency-based caching is caused by several factors:

- Smaller and simpler queries to an SMT solver are issued.
- Number of queries is reduced by caching.
- Number of queries is further reduced by the fact that the syntactic equality optimization can work even in sequential benchmarks when multi-states are split into sub-states.

In our test case, dependency-based caching issued 5 times less queries compared to SymDIVINE with no caching. We also managed to solve 48 more benchmarks with Partial store compared to SMT store within timeout 4 minutes.

Table 5.1: Summary results showing effects of Partial SMT store and caching for each benchmark category. Equal queries denote number of state comparisons, solver queries denote number of state comparisons that used SMT solver.

Category name	Time with SMT Store [s]	Time with Partial Store [s]	Percentage difference	Equal queries	Solver queries SMT store	Solver queries Partial store
bitvector	4821.3	2531.5	−47.5 %	165470	165470	6854
eca	488.9	696.1	42.4 %	18695	18695	5173
loops	81.5	58.3	−28.4 %	5755	5745	4113
locks	243.9	237.0	−2.8 %	2040	2040	377
recursive	16.0	16.6	4.0 %	372	372	109
ssh-simplified	5172.0	3069.5	−40.7 %	238280	238280	571
systemc	5059.0	1920.9	−62.0 %	225906	225906	43513
concurrency	943.0	494.6	−47.6 %	1769313	96214	9619
ltl	297.4	297.7	0.1 %	14888	14854	0
summary	17 420.5	9620.0	−44.8 %	2455607	782430	140658

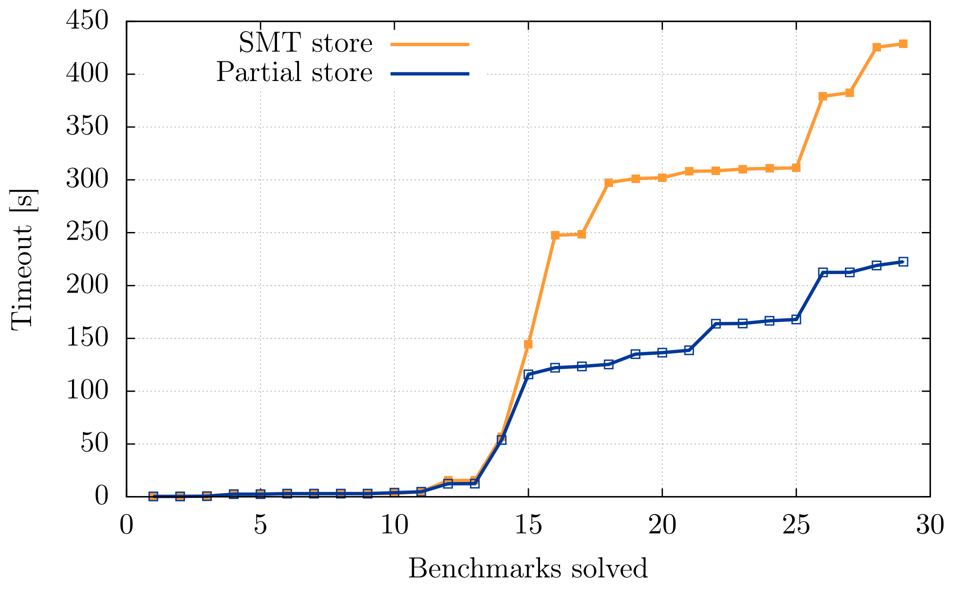

Figure 5.1: Effect of caching on bitvector benchmark set. Diagram depicts how many benchmarks could be solved within given time-out.

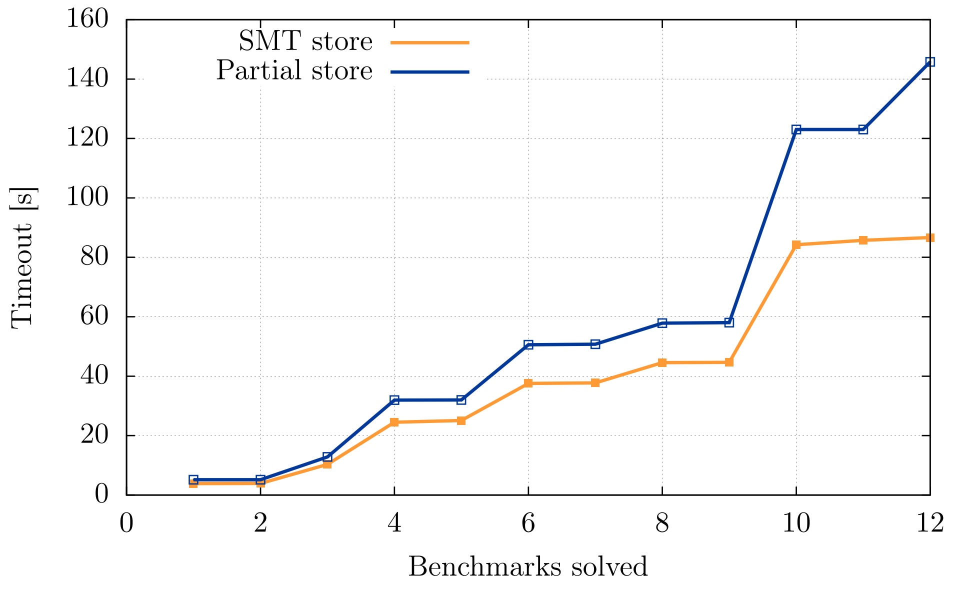

Figure 5.2: Effect of caching on eca benchmark set. Diagram depicts how many benchmarks could be solved within given time-out.

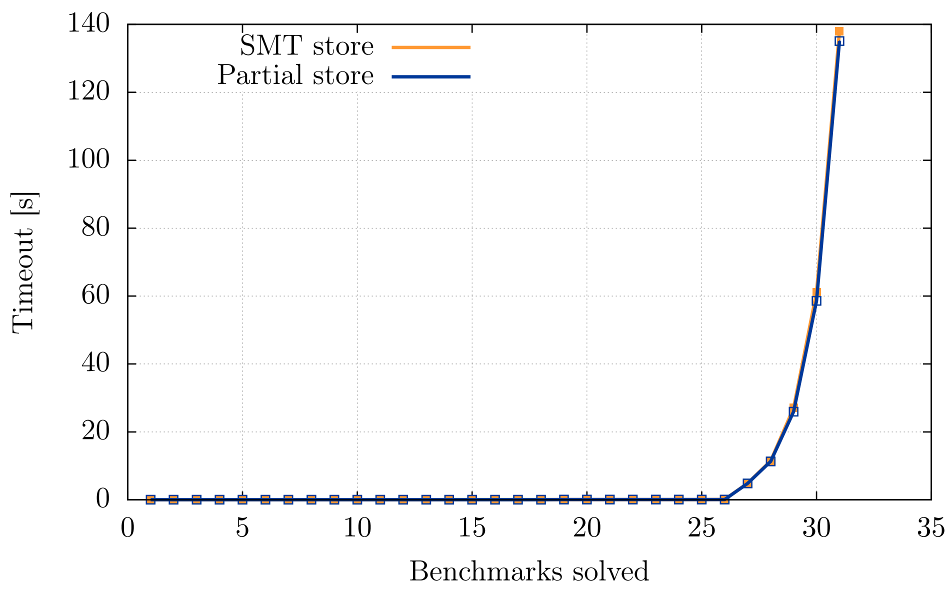

Figure 5.3: Effect of caching on locks benchmark set. Diagram depicts how many benchmarks could be solved within given time-out.

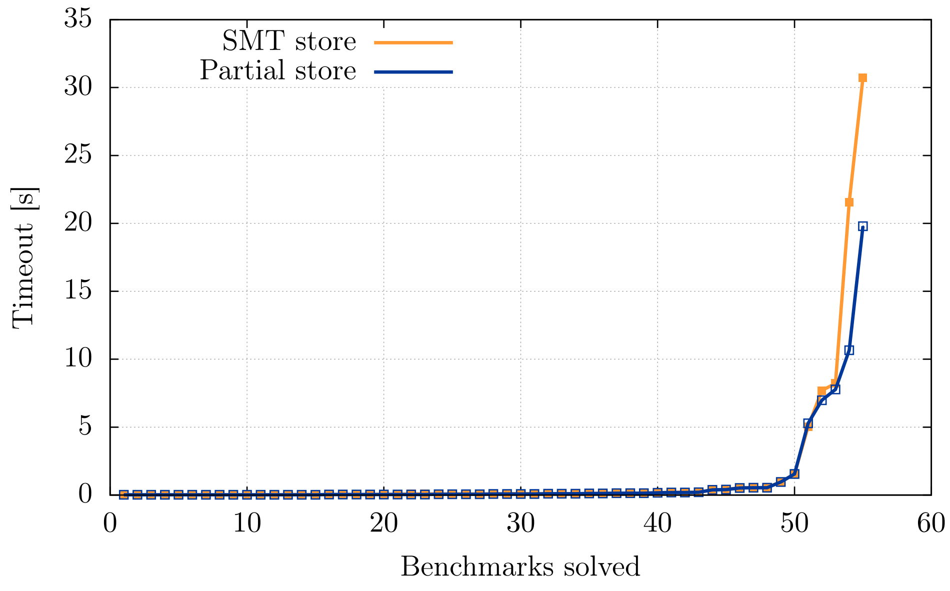

Figure 5.4: Effect of caching on loops benchmark set. Diagram depicts how many benchmarks could be solved within given time-out.

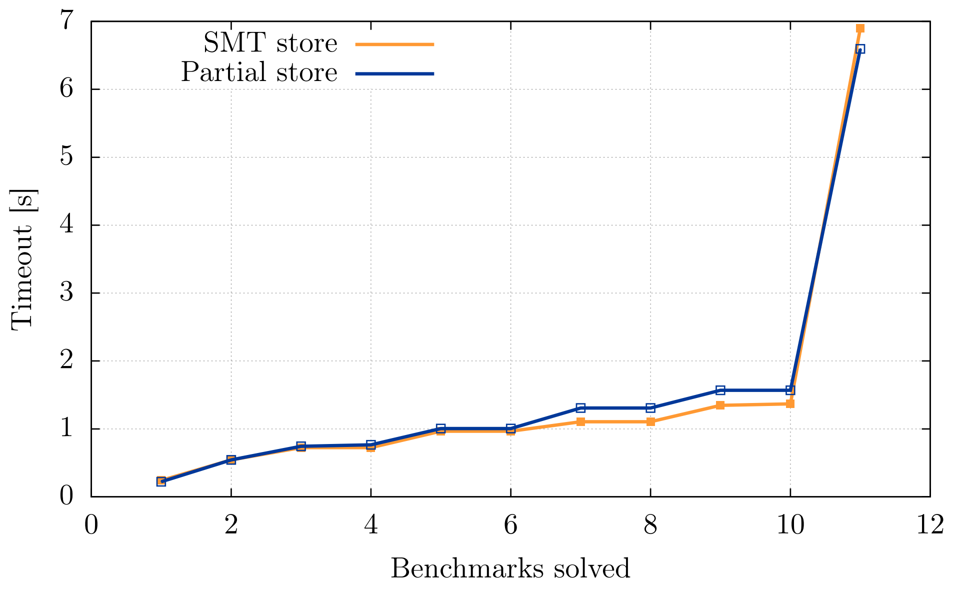

Figure 5.5: Effect of caching on recursive benchmark set. Diagram depicts how many benchmarks could be solved within given time-out.

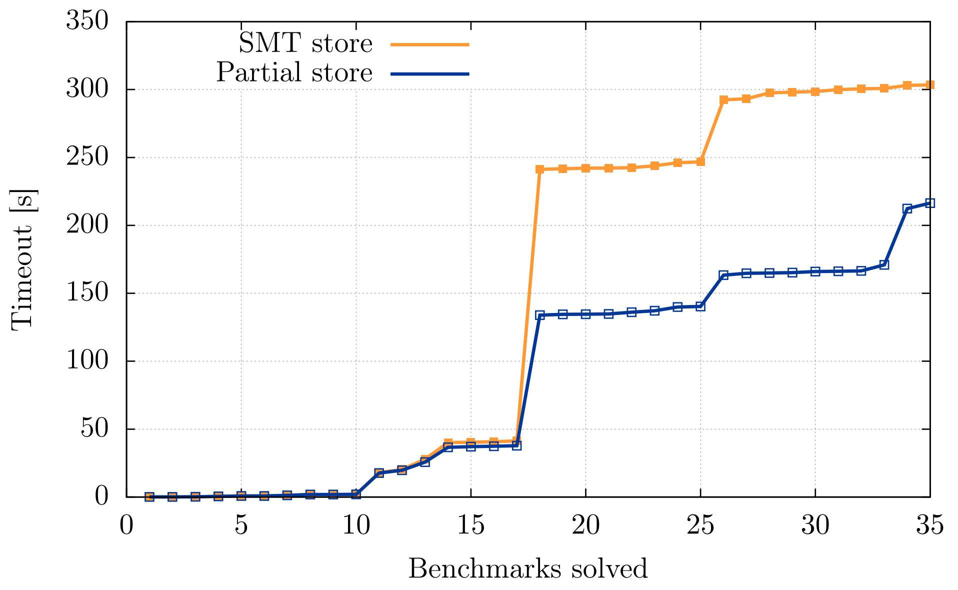

Figure 5.6: Effect of caching on ssh-simplified benchmark set. Diagram depicts how many benchmarks could be solved within given time-out.

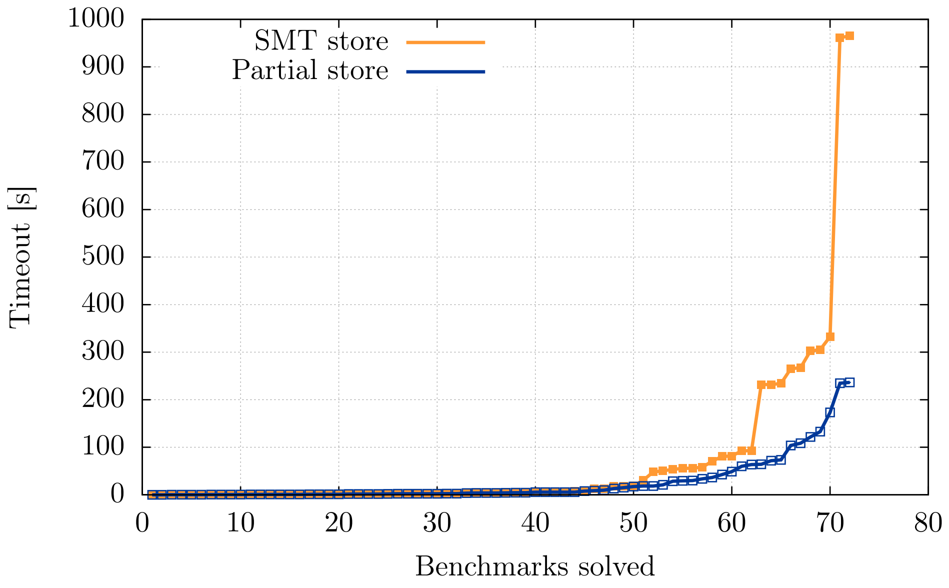

Figure 5.7: Effect of caching on systemc benchmark set. Diagram depicts how many benchmarks could be solved within given time-out.

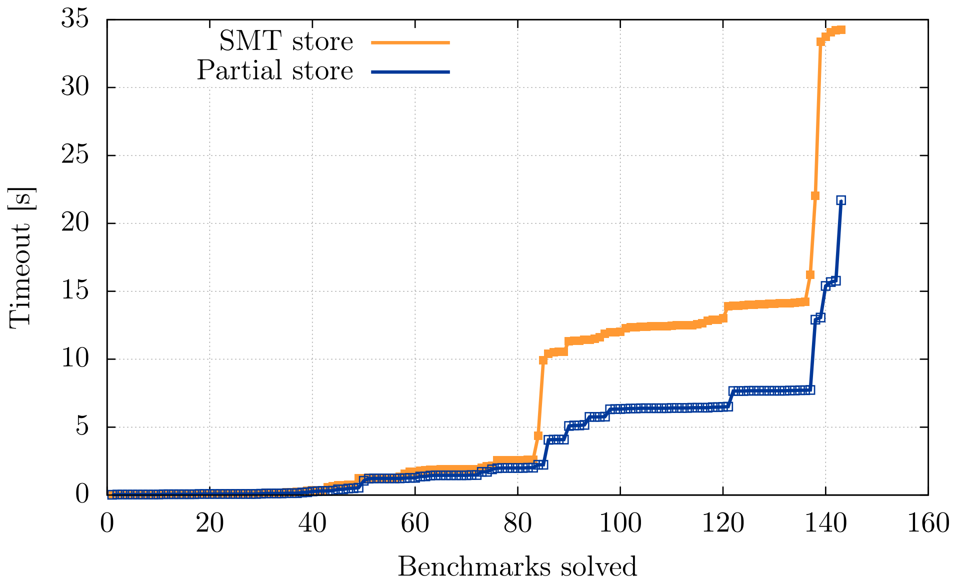

Figure 5.8: Effect of caching on concurrency benchmark set. Diagram depicts how many benchmarks could be solved within given time-out.

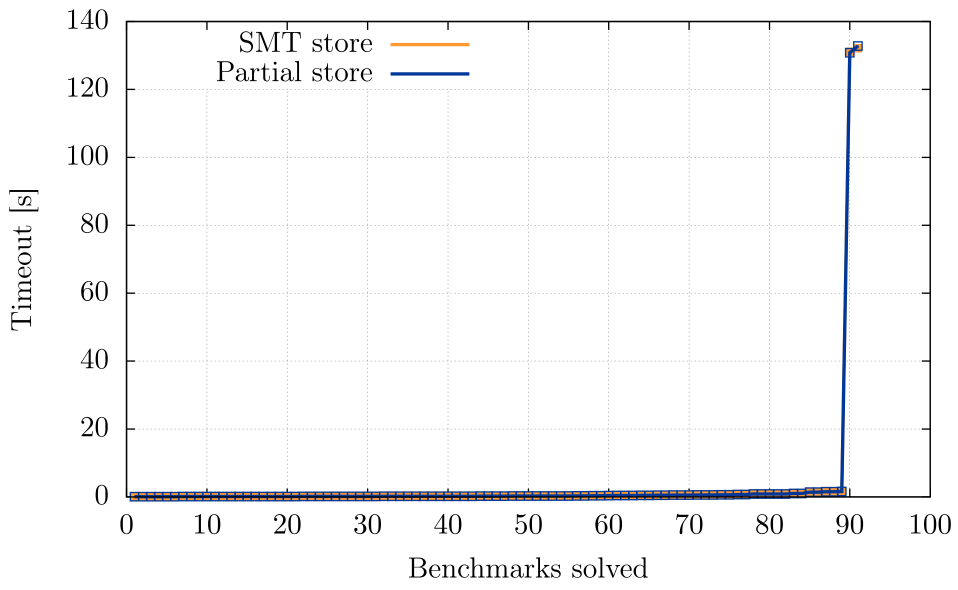

Figure 5.9: Effect of caching on LTL benchmark set. Diagram depicts how many benchmarks could be solved within given time-out.

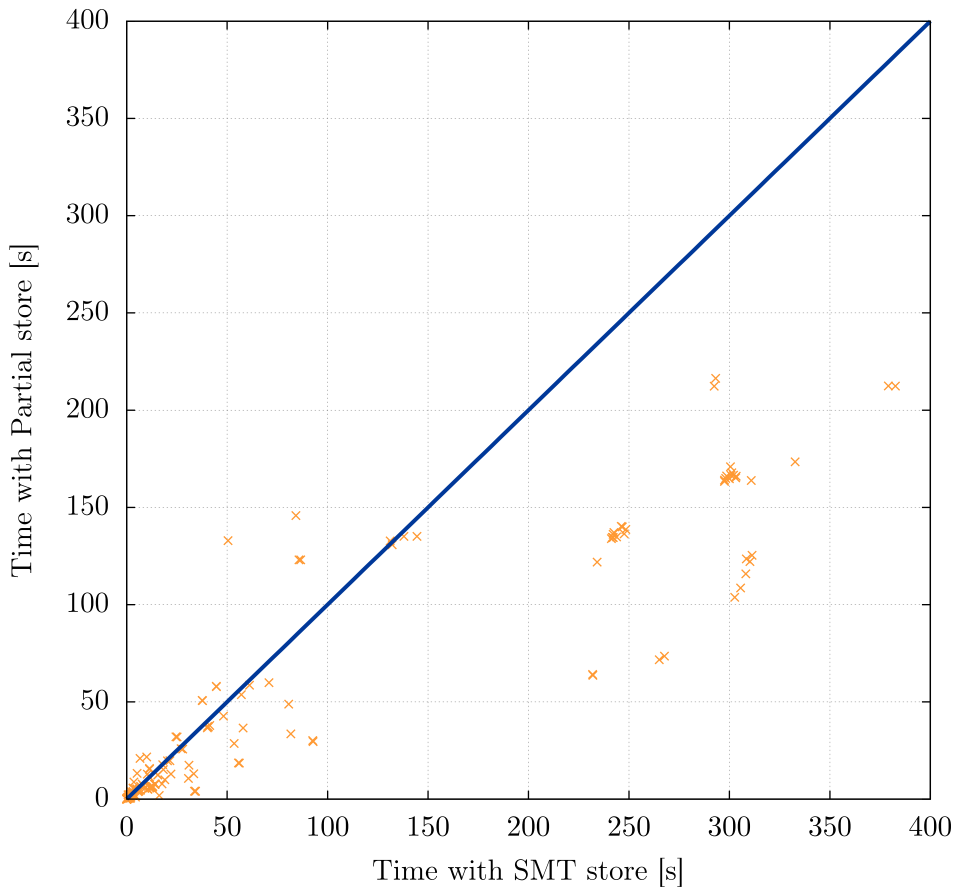

Figure 5.10: Overall summary of caching effect. Each point represents a single benchmark. Benchmarks under the blue line are the ones, which can be verified faster using caching.

Chapter 6

Conclusion

We showed that caching of queries to an SMT solver is a possible way of speeding-up control-explicit data-symbolic model checkers. To do that, we proposed *dependency-based* caching, a method for decomposition of quantified SMT queries and follow-up memorization of its results, and performed evaluation of our implementation in the SymDIVINE model checker.

Decomposition of queries in dependency-based caching takes inspiration from classical caching algorithms for quantifier-free SMT queries used in symbolic execution, and extends the core idea to quantified queries in control-explicit data-symbolic model checking while taking advantage of the way it is constructed in these tools. It works by computing data dependencies among variables and splitting a single multi-state into multiple so-called sub-states according to the dependencies. This allows to split a single large SMT query into multiple smaller ones, whose satisfiability results can be memorized, and therefore prevent issuing the same query again.

Dependency-based caching was implemented in SymDIVINE and evaluated using SV-COMP benchmarks for reachability and several other benchmarks for LTL model checking. The evaluation shows that caching can save a large amount of verification time for large benchmarks (in our experiments up to 75 %) and therefore bring verification of real-world sized parallel programs closer to reality. Our experiments show that, in summary, verification can be two times faster using dependency-based caching.

While working on this thesis, we improved usability and performance of LTL model checking algorithm in SymDIVINE. We have also uncovered a bug in the implementation, which made some runs of a verified program infeasible and therefore SymDIVINE could provide false negative results in some cases.

6.1 Future Work

In the future, we would like to make a more detailed examination of SMT queries in SymDIVINE. We would like to experimentally evaluate if issuing queries as small as possible leads to an optimal performance of an SMT solver or if we could benefit from issuing large queries produced by merging several sub-states together.

SymDIVINE currently supports only Z3 SMT solver. It would be interesting to implement support for other solvers and compare the effect of caching. We would also like to explore the possibilities of optimizations like purely syntactic equality of state, as we have seen that these optimizations can be more effective when issued on sub-states instead of multi-states.

Finally, there is a potential in canonization of control-flow location representation, as we realised during the development of caching techniques. In the current version of SymDIVINE, threads are identified by the order in which they are created. This can lead to a production of two multi-states with different thread naming and the same data valuation. However, due to the naming, they are considered to be different. Canonization of control-flow location representation would lead to merging such two states and therefore to the reduction of the multi-state space.

Bibliography

[1] Hans van Maaren Armin Biere Marijn Heule and Toby Walsh. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. ISBN: 978-1-58603-929-5.

[2] Jiří Barnat, Petr Bauch, and Vojtěch Havel. “Model Checking Parallel Programs with Inputs”. In: *Parallel, Distributed and Network-Based Processing*. 2014. DOI: 10.1109/PDP.2014.44.

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

[4] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. “CVC4”. In: *Computer Aided Verification*. Springer, 2011. DOI: 10.1007/978-3-642-22110-1_14.

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.

[6] P. Bauch, V. Havel, and J. Barnat. “LTL Model Checking of LLVM Bitcode with Symbolic Data”. In: *Mathematical and Engineering Methods in Computer Science*. Springer, 2014. DOI: 10.1007/978-3-319-14896-0_5.

[7] Dirk Beyer. “Software Verification and Verifiable Witnesses”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015. DOI: 10.1007/978-3-662-46681-0_31.

[8] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. “The OpenSMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2010. DOI: 10.1007/978-3-642-12002-2_12.

[9] Randal E. Bryant. “Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams”. In: *ACM Computing Surveys*. Association for Computing Machinery, 1992. DOI: 10.1145/136035.136043.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs". In: *USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2008. URL: `http://dl.acm.org/citation.cfm?id=1855741.1855756`.

[11] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. "The nuXmv Symbolic Model Checker". In: *Computer Aided Verification*. Springer, 2014. DOI: `10.1007/978-3-319-08867-9_22`.

[12] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999. ISBN: 0-262-03270-8.

[13] Vojtěch Havel. "Generic Platform for Explicit-Symbolic Verification". Master Thesis. Masaryk University, Faculty of Informatics, Brno, 2014. URL: `http://is.muni.cz/th/359437/fi_m/`.

[14] Gerard J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on Software Engineering*. Institute of Electrical & Electronics Engineers (IEEE), 1997. DOI: `10.1109/32.588521`.

[15] 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)*. Institute of Electrical & Electronics Engineers (IEEE), 1995.

[16] James C. King. "Symbolic Execution and Program Testing". In: *Communications of the ACM*. Association for Computing Machinery, 1976. DOI: `10.1145/360248.360252`.

[17] Daniel Kroening and Michael Tautschnig. "CBMC – C Bounded Model Checker". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014. DOI: `10.1007/978-3-642-54862-8_26`.

[18] Chris Lattner. *The LLVM Compiler Infrastructure Project*. 2016. URL: `http://llvm.org/` (visited on 05/09/2016).

[19] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008. DOI: `10.1007/978-3-540-78800-3_24`.

[20] Jan Mrázek, Petr Bauch, Henrich Lauko, and Jiří Barnat. "SymDIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration". In: *Symposium on Model Checking of Software*. Springer, 2016. DOI: `10.1007/978-3-319-32582-8_14`.

[21] Hristina Palikareva and Cristian Cadar. "Multi-solver Support in Symbolic Execution". In: *Computer Aided Verification*. Springer, 2013. DOI: `10.1007/978-3-642-39799-8_3`.

[22] LLVM Project. *LLVM Language Reference Manual*. 2016. URL: `http://llvm.org/docs/LangRef.html` (visited on 05/09/2016).

[23] Petr Ročkai, Jiří Barnat, and Luboš Brim. "Improved State Space Reductions for LTL Model Checking of C & C++ Programs". In: *NASA Formal Methods*. Vol. 7871. Springer, 2013.

[24] Nikolai Tillmann and Jonathan de Halleux. "Pex: White Box Test Generation for .NET". In: *Tests and Proofs*. Springer, 2008. DOI: `10.1007/978-3-540-79124-9_10`.

[25] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. "Green: Reducing, Reusing and Recycling Constraints in Program Analysis". In: *Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2012. DOI: `10.1145/2393596.2393665`.

[26] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. "SPASS Version 3.5". In: *Automated Deduction*. Springer, 2009. DOI: `10.1007/978-3-642-02959-2_10`.

[27] Josef Weidendorfer. *Callgrind, Part of Valgrind Project*. 2016. URL: `http://valgrind.org/` (visited on 05/09/2016).

Appendix A

Archive Structure and Running SymDIVINE

A.1 Archive Structure

The archive submitted with this thesis contains a snapshot of the git repository[1] with SymDIVINE source code and sources of the thesis itself with all the measurements we have used to evaluate dependency-based caching.

The repository maps whole development of SymDIVINE during writing of this thesis including re-implementation of LTL algorithm, extension of LLVM interpreter to support two new instructions, implementation of naive caching, implementation of dependency-based caching and several bug-fixes and small improvements of the tool.

A.2 Running SymDIVINE

SymDIVINE can be either compiled from source or obtained in binary form repository release page[2]. Runtime dependencies of SymDIVINE are Z3 and ltl2tgb, libboost-graph1.54.0, gcc-4.9 (or higher) and g++-4.9 (or higher).

To compile SymDIVINE, following dependencies has to be installed on the target system: CMake 2.8 (or higher), make, llvm-3.4, boost, flex and bison. SymDIVINE then can be compiled by:

```
./configure
cd build
make
```

Final binary file is located in the `bin` directory. SymDIVINE then can be run using following commands (first one runs reachability algorithm, the second one run LTL algorithm):

[1] https://github.com/yaqwsx/SymDivine

[2] https://github.com/yaqwsx/SymDIVINE/releases

```
bin/symdivine reachability <model.ll> [options]
bin/symdivine ltl <property> <model.ll> [options]
```

See `bin/symdivne help` for more info. It is also possible to use helper script `scripts/run_symdivine.py`, which takes a C a C++ file, compiles it and runs SymDIVINE, to easily start verification of a C or C++ benchmark.

Appendix B

Measurements

All our measured and processed data can be found in the electronic archive in the `experiment_results` directory.