Addressing the Performance Bottlenecks of Device Driver Synthesis

Alexander Legg

Supervised by
Leonid Ryzhyk, Nina Narodytska, and Gernot Heiser

June 3, 2016

Contents

Abstract

Device drivers are notorious for their contribution to the failure of software systems. A driver forms the interface between two complicated systems: an operating system and a hardware device. As a result, driver development is complicated and highly prone to human error. Driver reliability has been the focus of much research over the past two decades. One result of this research is device driver synthesis.

Device driver synthesis aims to remove human error from the development process by removing humans. A synthesis tool can take a formal specification of requirements and produce a system. Thus, a driver can be synthesised by providing the tool with a specification of the OS and the hardware that it must interface with. The resulting driver is guaranteed to be correct with respect to the specifications.

The drawback of this approach is that synthesis is a computationally intensive task belonging to 2-EXPTIME. In practice synthesis tools are capable of producing drivers though there remain cases where synthesis is impossible. In this thesis I will present an algorithm that provides results on many of these previously unattainable cases.

Acknowledgements

Acknowledge some people

1 Introduction

This is the introduction.

1.1 Device Drivers

Device drivers are the software that allows the operating system to interface with hardware. The role of the driver is to manipulate the inputs of the device so that it remains in a error-free state and correctly handles the requests of the operating system. By way of example consider an ethernet driver. The driver accepts requests to send and receive data packets from the OS and acts on those requests by reading from and writing to buffers on the device. It must ensure that those buffers are maintained in a usable state by correctly updating a register containing the location of the head of the buffer queue.

According to a study performed in 2011 [10], drivers account for approximately 57% of the lines of code in the Linux kernel and subsequently is the largest source of bugs. The study also analysed the staging directory of the kernel, which contains all in-progress drivers, and found it to contain the highest fault rate (faults per line of code) out of any directory in the kernel. The results of the study give evidence to the widely held belief that correct drivers are hard to produce.

Consequences of buggy drivers

This thesis focuses on automatic construction of correct drivers as a solution to the driver problem. Alternate approaches, of which there are many, will be discussed in Chapter 2.

1.2 Synthesis

Explain what synthesis is

Briefly describe the current state of the art

Explain state space explosion

1.3 Synthesis with SAT

Explain SAT solvers

Explain how they might help with state space explosion

(Explain that there is room for both approaches in the world)

Explain QBF and/or why this is nontrivial

2 Related Work

3 Background

Synthesis is a process that demands mathematical formalisation in order to provide a strong guarantee of the correctness of the resultant software. As such we require a mathematical language to describe the system we wish to produce, the environment in which it operates, and the properties we want the system to adhere to. This chapter will outline that language and the ways we can reason about what we describe in it.

3.1 Temporal Logic

Devices drivers are an example of a reactive system. A reactive system is a system that is in a continuous process of responding to input from its environment. A driver accepts requests from the operating system and state information from the device, and it responds by sending commands to the device and reporting to the operating system.

In an ordinary program, correctness can be ascertained by observing the final output. A reactive system does not terminate and is only considered correct if it adheres to its specification indefinitely. A formalism of a reactive system must then consider the concept of time in order to specify its correctness property.

3.1.1 Kripke Structures

A reactive system can be thought of as a sequence of *states*. The system *transitions* between these states as it responds to its inputs. A Kripke structure [8] formalises this notion and provides us will the language to reason about a reactive system.

A Kripe structure M is defined by the tuple $M = (S, S_0, R, L)$ with respect to a set of atomic propositions AP.

- A finite set of states, S,
- a subset of initial stats $S_0 \subseteq S$,
- a transition relation $R \subseteq S \times S$, and
- a labelling function $L : S \to 2^{AP}$.

The transition relation defines how the system moves between states. It must be left-total, i.e. for every $s \in S$ there is an $s' \in S$ s.t. $R(s, s')$. The labelling function maps every state in S to a set of atomic propositions that hold in that state of the system.

We often consider *paths* or *runs* of a Kripke structure. A path is a sequence of states $\pi = s_0, s_1, s_2, ...$ such that $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

3.1.2 Linear Temporal Logic

Kripke structures lay the groundwork for reasoning about reactive systems. Using the labelling function we may define desirable properties for the system. What is now lacking is a means of bringing states together to reason about the system as a whole. This requires a logical language that can express temporal properties.

Temporal logic takes propositional logic and provides additional semantics for the concept of time. In a Kripke structure this refers to the expressiveness to reason about runs of the system. This allows us to define properties that must be true for the entire execution of a reactive system.

Linear temporal logic (LTL) allows for statements that refer to a single run of a Kripke structure. Pnueli introduced LTL in 1977 [12] to succinctly describe the outcomes of program execution by referring to global invariants and eventualities. The syntax is:

- ϕ is a propositional formula referring to the current state,
- $X\phi$ - ϕ is true in the next state of the execution,
- $F\phi$ - Eventually (finally) ϕ will be true, and
- $G\phi$ - ϕ is always (globally) true.

- $\phi_1 U \phi_2$ - ϕ_1 holds until ϕ_2 holds.

These operators are semantically defined with respect to a Kripke structure $M = (S, S_0, R, L)$. We use $M, s \models \phi$ to denote ϕ holds true at state $s \in S$ of structure M. We definte $\models$ recursively:

- $M, s \models \phi$ iff $\phi \in L(s)$.
- $M, s \models \neg\phi$ iff not $(M, s \models \phi)$.
- $M, s \models \phi_1 \wedge \phi_2$ iff $(M, s \models \phi_1) \wedge (M, s \models \phi_2)$.
- $M, s \models \phi_1 \vee \phi_2$ iff $(M, s \models \phi_1) \vee (M, s \models \phi_2)$.
- $M, s \models X\phi$ iff for some state s', $R(s, s') \wedge M, s' \models \phi$.
- $M, s_0 \models F\phi$ iff for some path $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi))$.
- $M, s_0 \models G\phi$ iff for some path $(s_0, s_1, ...)$, $\forall i(i \geq 0 \wedge (M, s_i \models \phi))$.
- $M, s_0 \models \phi_1 U \phi_2$ iff for some path $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi_2) \wedge \forall j(j \geq 0 \wedge g < i \rightarrow (M, s_j \models \phi_1)))$.

Throughout this thesis will use F and G to represent the *finally* and *globally* operators. Elsewhere in the literature $\Diamond$ and $\Box$ are sometimes used to represent the same.

3.1.3 Computation Tree Logic

In addition to LTL, which is used to formalise properties about a single execution trace, we may need the ability to talk about aggregations of traces. In 1981 Clarke introduced computation tree logic (CTL) [5], which has additional syntax and semantics for exactly that. The syntax of CTL is as follows:

- $A\phi$ - ϕ is true on all paths
- $E\phi$ - there exists a path on which p is true

We again define the semantics of CTL with respect to a Kripke structure $M = (S, S_0, R, L)$.

- $M, s \models \phi$ iff $\phi \in L(s)$.

- $M, s \models \neg\phi$ iff $\neg(M, s \models \phi)$.
- $M, s \models \phi_1 \wedge \phi_2$ iff $(M, s \models \phi_1) \wedge (M, s \models \phi_2)$.
- $M, s \models \phi_1 \vee \phi_2$ iff $(M, s \models \phi_1) \vee (M, s \models \phi_2)$.
- $M, s \models EX\phi$ iff for some state s', $R(s, s') \wedge M, s' \models \phi$.
- $M, s \models AX\phi$ iff for all states s', $R(s, s') \rightarrow M, s' \models \phi$.
- $M, s_0 \models A[\phi_1 U \phi_2]$ iff for all paths $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi_2) \wedge \forall j(j \geq 0 \wedge g < i \rightarrow (M, s_j \models \phi_1)))$.
- $M, s_0 \models E[\phi_1 U \phi_2]$ iff for some path $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi_2) \wedge \forall j(j \geq 0 \wedge g < i \rightarrow (M, s_j \models \phi_1)))$.
- $(M, s \models AF\phi) \Leftrightarrow (M, s \models A[\top U \phi])$
- $(M, s \models EF\phi) \Leftrightarrow (M, s \models E[\top U \phi])$
- $(M, s \models AG\phi) \Leftrightarrow (M, s \models \neg EF(\neg\phi))$
- $(M, s \models EG\phi) \Leftrightarrow (M, s \models \neg AF(\neg\phi))$

In CTL, each A or E must be paired with an LTL operator. For example $AG\phi$, which says that ϕ must always hold on all paths. Alternatively, CTL* allows for free mixing of operators from LTL and CTL. This allows for terms such as $E(GF\phi)$, which is true iff there exists a path where ϕ will always be true at some future state.

3.2 Model Checking

Once the desirable properties of a system have be laid out in temporal logic, the next step is to verify that a program adheres to its specification. One approach is to manually (or with the assistance of a proof checker) construct a proof for the program. However, this costs the verifier in both time and mental effort and so a mechanised approach is desired. The first such automatic *model checker* was proposed by Clarke et al. [4] to verify temporal properties of finite state programs. The algorithm they proposed was a search based labelling of a finite state-transition graph, representing the program, with subformulas of the temporal logic specification.

Another approach is based on the notion that temporal logic properties can be expressed in terms of automata theory. Specifically, a finite state automaton over infinite words can be used to represent a temporal logic formula. Büchi automata [3] are ω-automata, i.e. finite automata that accept an infinite stream of input, which may be constructed such that the automaton will accept exactly the inputs allowable by a temporal logic formula.

In [13], the authors propose a model checking approach using this connection between temporal logic and automata theory. They propose the construction of a finite state, infinite word generator representing the program P, and an acceptor of the same, ϕ, constructed from the temporal property to be checked. Thus the program may be model checked by determining whether $P \cap \neg\phi$ is empty.

3.2.1 Büchi Automata

Like all ω-automata, the language of a Büchi automaton is ω-regular, i.e. a regular language extended to infinite streams. A regular language over the alphabet Σ is

- The empty language , *or*
- A singleton language $\{a\}$ for $a \in \Sigma$, *or*
- For two regular languages A and B:
  - $A \cup B$ the union of those languages, *or*
  - $A \,\textbf{.}\, B$ the concatenation of those languages, *or*
  - $A*$ the Kleene operation on that language.

The automata itself is defined as a tuple $A = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta : Q \times \Sigma \to Q$ is a transition function mapping states and letters to next states,
- $q_0 \in Q$ is an initial state, and
- $F \subseteq Q$ is a set of accepting states. A accepts an input stream iff it visits F infinitely often.

3.3 Synthesis

Model checking is the art of deciding whether a program meets a specification. Synthesis is the related problem of building the program *to* a specification. In the model checking literature discussed thus far, the program and its environment are considered to be the same entity. This approach contains the assumption that the program and environment are cooperating, which is not appropriate for reactive synthesis. Consider a driver that must perform some action on input from the operating system. Driver synthesis with a cooperating OS could assume that the input never occurs. For synthesis of reactive programs an adversarial relationship more accurately formalises the problem. As such, the problem is reformulated as a two player game between the reactive program and its environment [11].

In this thesis we will consider *safety games* between a controller and the environment, which are game structures $G = (S,\mathcal{U},\mathcal{C},\delta,s_0,E)$ where

- S is a finite set of states,
- $\mathcal{U}$ is a finite alphabet of *uncontrollable* actions,
- $\mathcal{C}$ is a finite alphabet of *controllable* actions,
- $\delta : S \times \mathcal{U} \times \mathcal{C} \rightarrow S$ is a transition relation mapping states, uncontrollable actions, and controllable actions to next states,
- $s_0 \in S$ is an initial state, and
- $E \subseteq S$ is a set of *error* states.

Conceptually, the game structure is another finite state automaton where transitions are partially controlled by both players. In each state, the environment chooses an uncontrollable action from $\mathcal{U}$ and the system chooses a controllable action from $\mathcal{C}$. We consider only deterministic games where $\delta(s,u,c,s'_1) \wedge \delta(s,u,c,s'_2) \rightarrow (s'_1 = s'_2)$.

We modify the notion of a run to suit games: $(s_0,u_0,c_0),(s_1,u_1,c_1),...(s_n,u_n,c_n)$ where $\forall i[i \geq 0 \rightarrow \delta(s_i,u_i,c_i,s_{i+1})]$. We say that a run is *winning* for the controller iff it does not visit an error state. The game is zero-sum, therefore a run is winning for the environment in the dual case where an error state is visited.

Formally, a run is controller-winning iff $\forall i[i \geq 0 \rightarrow \neg E(s_i)]$ environment-winning iff $\exists i[i \geq 0 \wedge E(s_i)]$.

A *controller strategy* $\pi^c : \mathcal{S} \times \mathcal{U} \rightarrow \mathcal{C}$ is a mapping of states and uncontrollable inputs to controllable actions. A controller strategy is winning iff all runs $(s_0, u_0, \pi^c(s_0, u_0)), (s_1, u_1, \pi^c(s_1, u_1)) \ldots$ are winning. *Realisability* is the problem of determining the existence of a winning controller strategy.

An *environment strategy* $\pi^e : \mathcal{S} \rightarrow \mathcal{U}$ is a mapping of states to uncontrollable actions. An environment strategy is winning iff all runs $(s_0, \pi^e(s_0), c_0), (s_1, \pi^e(s_1), c_1) \ldots$ are winning for the environment. The existence of an winning environment strategy implies the nonexistence of a winning controller strategy and vice versa.

3.3.1 Symbolic Representation

The standard method to solve a safety game is to enumerate the sets of states that are winning for one of the players. In large systems this set may quickly become intractably large to explicitly enumerate, i.e. suffer from *state explosion*. An alternative, introduced by McMillan [9], is to symbolically represent a set of states.

3.3.2 Binary Decision Diagrams

Consider the game structure in the previous section. Without loss of generality, we may replace the set of states S with a set of boolean variables. A single state is now a valuation to those variables and a set of states may be symbolically represented by a boolean function. In order to combat state explosion a compact representation of boolean functions is required. The standard choice is an ordered binary decision diagram (BDD) [2].

BDDs represent boolean functions as directed acyclic graphs. Each node is contains a variable, each edge represents a decision (true or false) on its parent's variable. One node is designated root and each path through the graph will terminate in one of two sink nodes that represent whether the decisions on that path satisfy the boolean function or not. Isomorphic nodes (where both decisions lead to the same result) may be removed and isomorphic subgraphs may be merged in order to compress the function representation. The ordering of variables in the graph is important to this compressibility. In the worst case the representation is a tree with no removed or merged nodes, which is exponential in the number of variables. Given a boolean function and a variable ordering the corresponding BDD is canonical.

Conjunction and disjunction may be performed on BDDs via an algorithm with a running time of $O(n \times m)$ where n and m are the sizes of the two BDDs. The worst case of this algorithm is rarely reached however and in general the operation is efficient. Universal quantification may be performed by constructing the conjunction of the BDD with the quantified variable set one way and the BDD with the variable set the other way. Existential quantification works the same way with disjunction.

3.3.3 Solving Games

Given a safety specification how do we determine whether it is realisable or not? The standard symbolic game solving algorithm is centred on determining sets of states from which the controller can win. The building block of this algorithm is the uncontrollable predecessor (UPre), which returns a set of predecessor states from which the environment can force play into a given set. We define this operator for a game structure $G = (S, \mathcal{U}, \mathcal{C}, \delta, s_0, E)$ as

$$UPre(X) = \exists u \forall c \exists s \exists x [u \in \mathcal{U} \rightarrow (c \in \mathcal{C} \wedge s \in S \wedge x \in X \wedge \delta(s, u, c, x))]$$

To solve the game we iteratively build a set of states backwards from the error set using the uncontrollable predecessor. It is clear that this set will grow monotonically and (since the state space is finite) eventually converge on fixed point. We call this fixed point set the environment's winning set. If this set contains the initial state then the game is unrealisable and the environment's winning strategy is to always play to stay within its winning set. Conversely, if the initial state is outside the environment's winning set then the controller must have a strategy to avoid the error state and the specification is realisable. The winning set can be efficiently represented and computed with a BDD.

We will return to this basic algorithm in later chapters for a more thorough discussion.

3.3.4 State Explosion

3.4 SAT

The ability to prove existentially quantified boolean formulas satisfiable or unsatisfiable (SAT) is enormously useful for program verification. Significant research has led to many highly efficient solvers for the SAT problem. Modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [6,7]. This algorithm is a backtracking search that operates on the formula given in conjunctive normal form (CNF).

A CNF formula is a set of *clauses* of the form $(l_0 \vee l_1 \vee ... \vee l_n)$ where each *literal* l_i is a boolean variable or its negation. We call a clause with only one literal a *unit clause*. We call a variable *pure* if it appears in only one polarity in the formula. The DPLL algorithm propagates unit clauses and removes clauses with pure literals as its first step. Next a value is assigned to a variable and the algorithm recurses to search for a solution with that valuation. If none can be found then a backtracking occurs and the other polarity of the variable is tried. In modern solvers, clause learning is used to share information between different branches of the search tree. The algorithm terminates when the current variable assignment satisfies the formula, or the search space is exhausted.

3.4.1 Bounded Model Checking

In the previous section BDDs were discussed as a symbolic representation that compresses boolean functions and efficiently quantifies the function. A CNF representation of a function is not canonical and so does not necessarily suffer from the same state explosion problems as BDDs. Biere et al. [1] introduced a model checking methodology that takes advantage of CNF as a symbolic representation and utilises a SAT solver to efficiently operate on it. We discussed BDDs in the context of solving games, however the context of this section is model checking of CTL* properties. It suffices to say that BDDs are applicable to this area as well and have similar advantages and disadvantages.

Instead of constructing a set of winning states this technique, called *bounded model checking*, searches for runs of the game. Conceptually the BDD approach is similar to breadth first search and bounded model checking is similar to depth first search. In broad strokes, the new methodology consists of constructing a propositional formula representing the existence of a program trace of a certain length k that violates the specification. The formula is solved by a SAT solver, which returns either SAT: a counterexample to the specification, or UNSAT: there is no counterexample of length k. The algorithm executes this process for increasing lengths k, which we call the *bound*.

One difficulty of this approach is choosing a maximum bound that is both *sufficient* to verify the program correct and *feasible* to compute the result of the propositional formula. For a finite state automaton, the *diameter* is the minimal length such that every reachable state can by reached by a path of that length or less. The diameter is sufficient for model checking but not always feasible. In addition, computing the diameter itself is an inefficient quantified boolean formula. Despite this difficulty, bounded model checking is useful in many practical cases. In particular the ability to quickly find short counterexamples gives an advantage in cases when a BDD based approach hits state explosion issues.

3.5 Interpolation

4 Bounded Synthesis

4.1 Abstract Game Trees

4.2 Algorithm

4.3 Computational Learning

4.4 Optimisation: Strategy Shortening

5 Synthesis with SAT

6 Evaluation

7 Conclusion

Bibliography

[1] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[2] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[3] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science (Proc. 1960 Internat. Congr .)*, pages 1–11. Stanford Univ. Press, Stanford, Calif., 1962.

[4] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

[5] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs*, pages 52–71, 1981.

[6] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[7] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[8] Saul Kripke. Semantical considerations on modal logic. In *Acta Philosophica Fennica*, volume 16, pages 83–94. 1963.

[9] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[10] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 305–318, New York, NY, USA, 2011. ACM.

[11] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.

[12] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[13] Moshe Y. Vardi. *Logics for Concurrency: Structure versus Automata*, chapter An automata-theoretic approach to linear temporal logic, pages 238–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.